SNAPS

A Snapshot Allocator and Processor System
for use in a Distributed Database
with application to a
Radiology Database System

A thesis
submitted as partial fulfilment
of the requirements for the Degree
of
Masters of Science in Computer Science
in the
University of Canterbury
by

Richard Douglas Fish

University of Canterbury
1988

To my parents,
Nanette and Michael

Acknowledgements

I would like to thank my two supervisors. Firstly, Dick Cooper was the person who was involved initially with the project, and gave me the initial introduction to doctors Kennedy and Feltham, whom where wanting the system developed. He also gave me a lot of ideas that have become an integral part of the system. Neville Churcher became my supervisor from February 1987. He has been a guiding light since then, right through to the completion of my thesis. His support, encouragement, ideas, and advice on the presentation of the thesis have been invaluable.

I would like to thank Dr Kennedy, Head of the X-Ray Department of Christchurch Public Hospital, and Dr Feltham, Head of the X-Ray Department of Nelson Hospital for their information and advice on the requirements of the interconnection of the radiology systems.

I would like to thank my flatmates over the years last two years and my friends for encouraging me to carry on when times were not too good. I would also thank Chris Taylor for his advice on the presentation of this thesis.

Last but not least, I would like to thank my parents for their support through out my University years, and without them I would not have been able to complete my Masters Degree.

Contents

Abstract

Traditional snapshots have been proposed as a way of providing multiple copies of read only data at remote sites. There are limitations in this approach when this applied to distributed databases. The snapshots are read only. They are very specific to a user, and the data they capture.

The snapshot system I propose allows for snapshots to be updated. They are integrated into the local databases so that other users can access them as part of the local database, and no specifically access the snapshot.

Most approaches to distributed database design has involved integrating database in the same organisation. The system I propose is designed to be used with different organisations that are loosely joined. It is suitable for a database that is predominantly read only, and where limited update capability is required on only some relations in the database. It provides a high degree of autonomy to the different organisations. It is suitable when there is no large computer at each of the sites to co-ordinate the operations of the database.

This system was designed to be used in the radiology environment, where a number of different health care organisations wanted to be able to access patient x-ray records stored at other hospitals and clinics. Most of the data being stored is read only. The system was based on micro computers that were connected on a local area network.

Introduction

The 1970's and 1980's have been likened to the industrial revolution of the late 19th century. This era has seen the start of the information revolution. The 1970's was the decade when the computer came of age. From being an oddity in the universities, it became widely used in the commercial environment. The 1980's has seen the rise of the personal computer, the PC. The 1990's, I believe, is going to be the decade where telecommunications is going to revolutionise the way the world communicates. The information revolution looks set to accelerate with the advent of fast, efficient and cheap telecommunications which will be available to everyone.

Users have always wanted more and better information. Banks, airlines and hospitals, which have data stored over many sites, want to be able to access information at any one of them. However, the technology was not capable of providing this effectively and efficiently. This has changed with the combination of two very different technologies, database management systems and telecommunications, to give the user the most flexible system for doing this thus far developed : the distributed database management system.

The development of database management systems in the 1970's offered users a new way to store and retrieve information at one site. It allowed them a standardised way to store data and a flexible method of accessing and reporting.

At the same time, telecommunications were also beginning to become readily available. Local and wide area networks were being developed but were still very unreliable and expensive to operate.

Both areas of technology were very new in the 1970's. Each was going through the development stage, and though the idea of a distributed database system had been postulated, neither was capable or reliable enough to cope with the complex requirements of this type of system.

As telecommunication and database management systems became more accepted, and their performance and reliability were enhanced, distributed database systems became more feasible. The late 1970's and 1980's have seen the development and research into distributed database systems around the world.

A distributed database has parts of the database partitioned over one or more machines. It is designed to take advantage of the processing needs of its users and to utilize parallelism by distributing processing over the various machines in the network as much as possible. A full distributed database system should offer all the same facilities as a single site centralised database system, and should appear to the user to be a single site database. At present, all research into distributed databases is being done using the relational model [CODD70]. This is the model I shall use throughout this thesis.

An organisation that is geographically distributed has a good case for a distributed database system. A company with branches in a number of centres would be a good example. It will allow a manager the ability to have access to all this organisation's data, thus being able to make better decisions.

There is also a need for independent organisations to have access to the information that others have. For example, public and private hospitals, clinics, and other health care organisations hold large amounts of patient information, that would be valuable to all health professionals. The diversity of operations, and the sensitivity of medical information, have

meant that distributed database systems have not been fully utilized in this area.

The aim of this thesis was to design a database system that would connect a number of private and public hospital radiology departments and private x-ray clinics in Christchurch and a similar system for Nelson. The system has to provide flexible operation, but still provide secure access to information and provide a high degree of autonomy to the individual sites. This is important as all the organisations operate independently. [KENN85] and [FELT85].

An identical single site system was being installed at each site, and the radiologists wanted to be able to have on-line access to some of the information stored at other sites, such as x-ray records and patient details. The major requirement was for read only access to information, but update ability on selected parts of the database was also required. The ratio of enquiries to updates is very high. This is one of the important aspects and has a direct bearing on a number of decisions in the system design.

Snapshots are a selected portion of a database as of a particular time. In 1980, Abida and Lindsay first proposed system supported snapshots [ABID80]. This and another paper by Abida [ABID81], deal specifically with snapshots. In the two unpublished papers by Cooper [COOP86a] and [COOP86b]. In [COOP86a], the problems of maintaining snapshots, and when it is appropriate to use them, are discussed. The second paper [COOP86b], discusses the use of snapshots in MIMER. MIMER is a relational database management system.

In centralised database systems, snapshots provide an "as of" view of selected portions of the database to an application, without the complexity of having to execute the enquiry at that time. In the distributed database

environment, snapshots are seen as a way of providing read only multiple copies of part of the database, at individual sites. In the medical environment, large amounts of data are read only and snapshots would be appropriate, as most of the information stored in a hospital is of read only nature,(e.g. medical records).

Snapshots are read only in nature, and are stored as separate relations [ABID80]. In the general database system this method of implementation is appropriate, though there is room for deviation for specific applications. In this thesis, the system is designed to handle snapshots that can be updated, to allow for more flexible use of the data. Another deviation from the definition of snapshots as proposed by Abida and Lindsay, is that they are stored as part of the relations they are derived from, instead of having the snapshot stored in a separate relation.

In the system I propose there are two levels of snapshot. The top level would be a global snapshot of all the local databases, that would be stored separately in a separate database. This global snapshot would be maintained by a central controller. The second level is a snapshot of the global snapshot that would be stored as part of the local database. These levels are designed to simplify and speed up access, and provide a high degree of site autonomy. When a local site requests data from other sites it does not assume anything about where the data that will be returned comes from. It assumes that all the other sites are involved, when in reality the data being accessed has already been brought to one site and stored in the global snapshot.

Figure I.1 shows the structure of the system I propose. It includes an interface to the local database management system, called SNAPS : SNapshot Allocater and Processor System. SNAPS controls all the access between the site of the local database and the global snapshot, and creation

of local snapshots of the global snapshot. The global snapshot controller performs accesses, maintains locks, and manages the global snapshot.

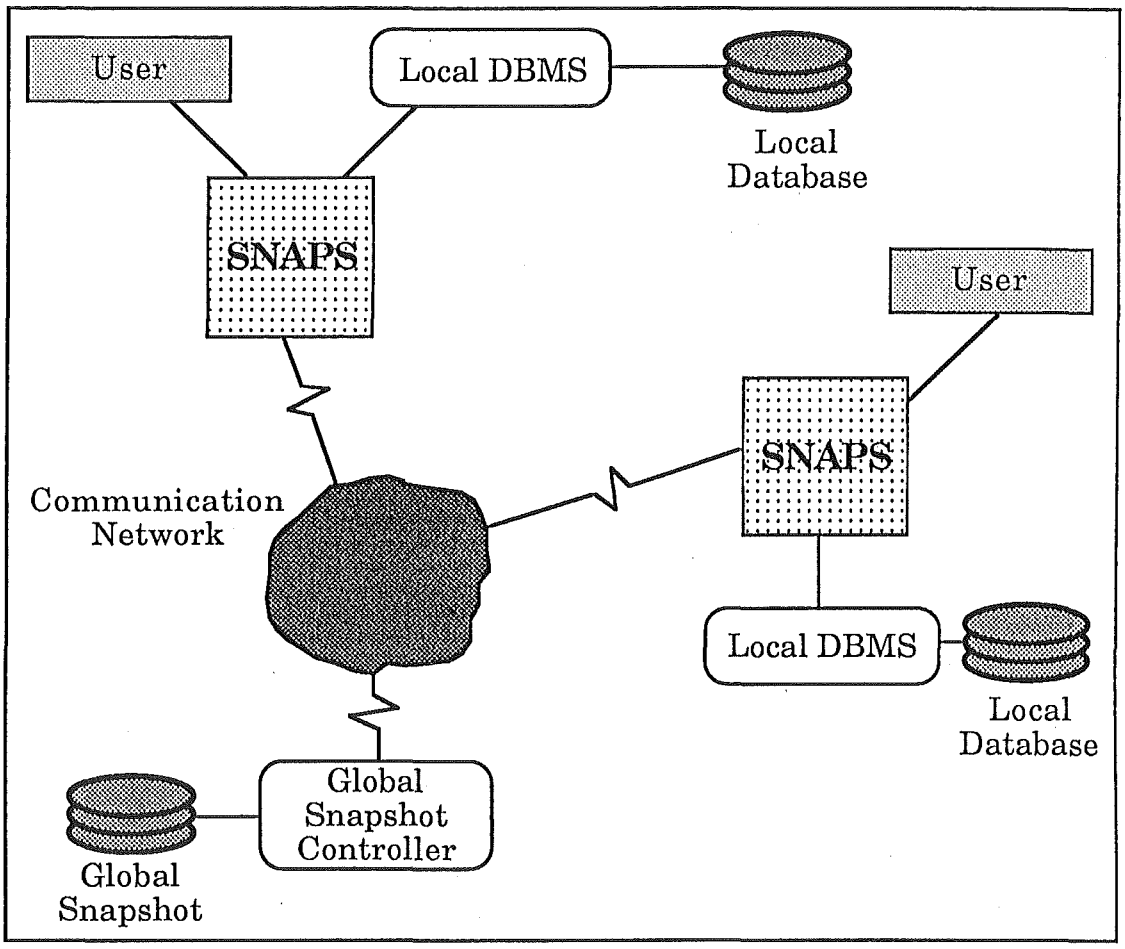

Figure I.1
The SNAPS Structure.

To provide a more flexible system, the ability to update these snapshots is proposed. One of the major arguments for snapshots is that they are read only, which makes them simple, and easy to maintain. The update mechanism that I propose, does not significantly increase the concurrency control costs required to maintain the snapshots above that of a read only query. The mechanism takes advantage of optimistic concurrency control [KUNG81]. Locks on a database object are granted to a site for the length of time between the global snapshot refresh. A site that wants to update part of the global snapshot only has to access the central site and does not know about the other sites. For this system, the global

snapshot would be refreshed once a day, at a time when usage of the system was low and communications charges were reduced.

The SNAPS system provides multiple copies using snapshots of the database. The system provides some features of multiple copies and some of snapshots. The snapshots are stored in the relations they are derived from when mapped onto the local databases, and the local snapshots can be updated. These updates do not alter the state of the global snapshot at the time they occur. Updates of local snapshots lock the corresponding tuples in the global snapshot, so that no other sites can not update them as well. The commitment of updates to local snapshots, and also additions and deletions to local relations, are only reflected in the global snapshot when the global snapshot is refreshed.

The system also provides features of snapshots in that the global snapshot is refreshed each day from modification to relations, and from changes made to the local snapshots. The local snapshots are refreshed automatically from the global snapshot, hence reflecting any changes in the global snapshot. To avoid the build up of local snapshots in the local database, any snapshot that has not been accessed at that site for a predetermined number of days is deleted from the local database, unless it has been explicitly placed into the local database permanently. This last feature is important, if other relations in the local databases are dependent on the local snapshot. For example, if values stored in a local snapshot of a patient record are being used by other relations such as the visit relation, then the patient record should not be deleted. This will be discussed more in Chapter 6.

To fully understand the complexity and requirements of a distributed database system, Chapter 1 introduces all aspects of distributed database systems. There are many researchers in the area of distributed databases.

Because of the large number of distinct areas within distributed databases, there have been few people who have taken an overall view of distributed databases, and most have presented surveys of current technology [CERI84] and [ULLM82]. In subsequent chapters, the areas identified in chapter 1 will be discussed in more detail, and in the context of the SNAPS system.

In Chapter 2, there is an introduction to the system requirements for the radiology system. It discusses the design and development of the single site system. The requirements of the interconnection of the radiology systems are introduced and possible solutions are discussed.

The concept of a global snapshot and local databases is introduced in Chapter 3. This chapter discusses how they will interact and the requirements of both. There appears to be no previous work done which has a global snapshot of parts of all the local databases, which is in turn accessed by the users at the local sites as a means of providing distributed access. There have been mentions in many papers about the possible use of snapshots to be used in distributed databases. In [ABID80], [COOP86a], [COOP86b], and [CERI84] possible uses are discussed, but their use is confined to a snapshot that is the result of a distributed query.

One of the biggest areas of research in distributed databases at the moment is query optimisation [APER83]. A query is any operation where data is accessed in a database. The main objective of most distributed query optimisation strategies is to reduce the amount of communication between sites. This is the approach I have taken, as the bandwidth of the wide area network the system is to use is low at 2,400 bits per second. The system I have design includes a number of optimisation techniques to improve availability and response time. The storing of a global snapshot in a separate database is one such optimisation. Because the site of the global

snapshot is the only other site a local site has to access the complexity of access and performing updates is greatly reduced. The storing of snapshots at the local sites in the relations that they are made up of is also an optimisation.

The main part of the thesis, database snapshots, are introduced in Chapter 6. The concept of the global snapshot, and the local snapshots will have been introduced in Chapter 4, and a general discussion of snapshot is in Chapter 2. This chapter discusses the extensions that I have introduced for the interconnection of the radiology systems. The concept of an updatable snapshot is introduced, with reference to the two levels of snapshot.

Having updatable snapshots leads to the problems of consistent transactions and concurrency control. The method I propose for the SNAPS system is optimistic concurrency control. This method does not attempt to acquire locks or timestamps for the data it wants to update, until it wants to commit the update [KUNG81]. The nature of the data being stored in the radiology system, and the unlikely case of having more than one site want to update data at the same time, means that optimistic control could be used to reduce communications and increase response time. In Chapter 6 the concurrency control method that is used in the SNAPS system is presented.

There are a number of algorithms that have been developed to implement the system. These are shown separately as they incorporate techniques from the previous chapters. This way the integration of the aspects of distributed databases used in the SNAPS system, discussed in the previous chapters, can be shown better. The algorithms shown perform a read only query on the global snapshot, perform an update of a local snapshot, refresh the global snapshot, and update the snapshots stored in

the local databases. In Chapter 7, the algorithms are shown in a Pascal type language, and an example of their operation is shown with reference to the radiology system.

A conclusion discusses the overall distributed system, and its potential for further development.

Chapter 1

Distributed Database Systems

"*A distributed database is a collection of data which is distributed over different computers of a computer network. Each site of the network has autonomous processing capability and can perform local applications. Each site also participates in the execution of at least one global application, which requires accessing data at several sites using a communication sub system.*" [CERI84].

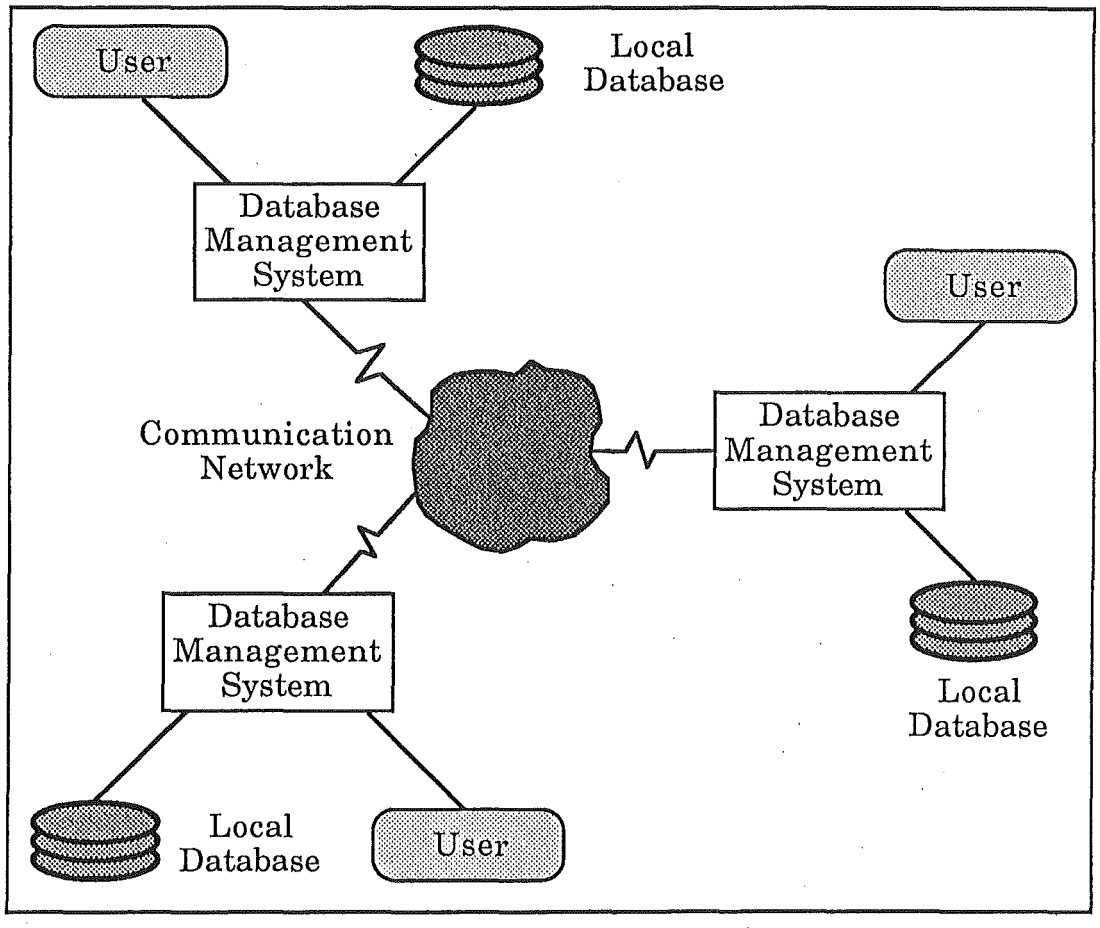

Figure 1.1
Architecture of a Generalised Distributed Database Management System

The definition and Figure 1.1 shows the two parts of a distributed database system : the integrated database, and the computer network. It identifies

one of the most important technological problems with distributed databases, the cooperation between autonomous sites.

Some of the major areas of a centralised database system are data independence, redundancy, efficient access, integrity, recovery, concurrency control, privacy, and security [CERI84]. These are all important issues in distributed database systems as well. Distributed database systems are not just centralised systems linked together. In this chapter, the overall design of distributed database system is discussed and compared to that of centralised database systems.

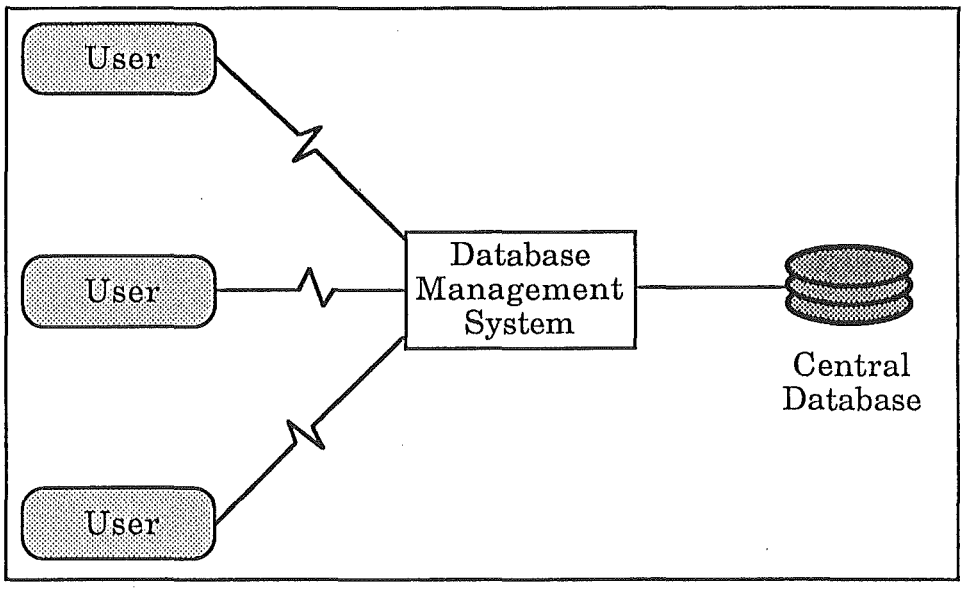

Figure 1.2
Architecture of a Central Database Management System

Figure 1.2 shows the structure of a centralised database management system. There are a number of users connected to a single database management system, which has only one database. It is different from the distributed database diagram, Figure 2, in that there is only one database, there is only one controller, and the users could connect to the database over a wide area network, or on a local network. In the distributed case all the users are connected to a local network or directly to their computer. This difference between the speeds of local and remote access is

significant and is generally orders of magnitude larger for local connections.

There are a number of advantages of a distributed database system over a centralised database system [APER83].

- increased availability

In a distributed database, the failure of a site or communication link does not necessarily imply that the whole database is inaccessible. If the data is stored redundantly, then the system is more resilient to certain types of failure. This is not the case in a centralised system. There is only one database and if the site where the database is stored fails or becomes partitioned from a site where the data is being accessed, then there is nothing that any site can do. It is unable to operate normally.

Because of the independent nature of the organisations that will use the radiology system, the interconnection system can not rely on all the sites being operative at one time, due to crashes, or the local systems being powered down. This is more likely than one large organisation. In a single organisation, there is likely to be a database administrator at each site, but in the x-ray departments or clinics there are no highly qualified database administrators that are capable enough to manage the database effectively. Therefore, there needs to be some sort of mechanism in the interconnection system that allows for the unpredictable nature of all the different organisations, which will ensure the availability of the database to all site within acceptable levels.

- decreased access time

If the data was all stored at one site, then every enquiry would have to access that site, which would require large numbers for messages and

transfer of data between the two sites. By partitioning and fragmenting the database over sites which are accessing those parts of the database frequently, the locality of the application can be increased and the access time will be significantly reduced. In the fully distributed environment, where each site communicates with all the other sites, there are still large amounts of communication required.

In the interconnection system I propose for the radiology environment, there is a compromise. There is a global snapshot, which holds selected portions of all the local databases, which is stored in a different database. Enquiries on the global snapshot can be requested by the local sites and the results stored in the local databases. Therefore the first time an enquiry is performed, the global snapshot would be accessed, but after that the data will be at the local sites, so that there would not be any need for any access to the central database.

The design of the radiology system, in Chapter 2, specifies the communication network requirements. The physical speed of the network is proposed as 2400 bits per second, which means the access time is very dependent on the communication time. Therefore to decrease access time the amount of communication has to be decreased, which is one of the important design specifications.

To reduce access time, data allocation and query optimisation are important aspects to consider. In Chapter 3, the question of data allocation is addressed, and in chapter 5, the whole question of query optimisation is introduced.

- easy expansion and possible integration of existing databases

There is a large investment in existing centralised database systems. Most of the early distributed database systems have been designed to take this

into account. A distributed system has been developed that acts as a front end to the local databases. This makes the systems easy to expand, as another site can be added easily without requiring any modification to existing databases.

The size of an organisations database will grow and it will eventually reach its physical limit. It will also be putting a large strain on the system and could case overloading of the computer. The upgrading of capacity will be easier for a distributed system than for a centralised one. A distributed system also reduces the system load by distributing the processing over a number of sites. With the availability of cheap and faster processing power it is not expensive to increase capacity as it used to be in extending a mainframe system.

This is an important aspect of the radiology system. The system has been designed to cope with new sites being added to the network, without interfering with other sites. At present the system is designed to work with only with the single site system that has been developed. In the future, organisations that already have a different computer system might be added to the system. The mapping mechanism proposed in Chapter 3 is designed to allow for the integration of similar existing databases. Discussions have been held with the Health Department about the possible integration of their database in the future. There is more discussion on the question of integration in Chapter 3.

Site Autonomy

Site autonomy is one of the most important aspects of a distributed database. In a centralised system there are no problems with site autonomy, as there are no other sites. In a distributed database system there is a hierarchical control structure, which includes a central database administrator who has the responsibility for the database as a

whole, and local database administrators who are responsible for their respective local database.

A local database administrator may have a high degree of site autonomy, up to the point that there is no central database administrator at all. In this case the local database administrators would have to act as controllers and co-ordinate all activities themselves. This case offers a high level of site autonomy, but puts a lot of reliance on the local database administrators to co-ordinate the operations. This has the draw back that there would be a large amount of inter-site communication needed to do this. An advantage however is that if a site crashes, it would have minimal impact on the system, as the other sites should be able to carry on operating.

At the other end of the scale, the central database administrator has almost complete control over the local database administrators. In this case the database would operate more like a central database, with the benefit of having centralised control but being very susceptible to the site of central database administrator becoming unavailable [CERI84].

This is an important area in the radiology system, as the sites in the network are independent organisations and no other site should have the ability to control the operations at any other site. This is important as in a commercial environment computer usage costs money.

Data Independence

Data independence is essentially that the organisation of the data is transparent to the application program. The program is written having a conceptual view of the data, and the main advantage of this is that any change to the physical organisation of the data does not affect the program. In a distributed database this adds a new aspect to data

independence : distributed transparency. This means that the database appear to be a normal central database, even though the data is distributed. [CERI84] and [ULLM82] give detailed descriptions of distributed transparency, and data independence.

One of the aims of the radiology system is to provide distributed transparency to all the users. The question of mapping and transparency are covered in more detail in Chapter 3.

Redundancy

The arguments for redundancy are the same for distributed databases and centralised databases. Redundancy in a centralised database is minimised as far as possible, to reduce the potential for inconsistency. Inconsistency among several copies of the same data item is avoided by only having one copy, which in turn reduces storage requirements. In distributed databases there are a number of reasons why redundancy is desirable. The speed of applications can be increased if the data is replicated at the sites where the applications need it. The availability of the system is increased, because if a site fails then it does not stop the execution of applications at other sites. If a site fails then another site can be used to get a copy of the data. [CERI84], [JOSE86], [PAPA84] and [ULLM82] all discuss the issue of replicated data.

In general, as the ratio of retrievals to updates increase, replication becomes more appropriate. It is more appropriate because if there are several copies, then any one of them can be used to satisfy a query, but all copies have to be kept up to date to ensure the database is consistent. Thus the data is more available, and the cost of updating is still low in comparison.

To increase availability in the radiology system, snapshots are used to provide multiple copies of the database. Snapshots are introduced in chapters 2 and how they are implemented in the SNAPS system in Chapter 6.

Efficient Access

One of the most integral parts of centralised systems is their complex accessing structure, such as secondary indexes and interfile chains. These structures provide for more efficient access to the data.

However in a distributed system, the difficulty of building and costs of maintaining such structures is too high in terms of communications costs. They are unsuitable for use in a distributed database, as these methods work at the tuple level in a database. This is a very inefficient way of working in a distributed system [CERI84].

For example consider the following schema and following query which retrieves all the patients for the doctor "Feltham".

```
doctor  ( name , surgery )
patient ( name , doctor-name , address , age )

retrieve (patient.name)
where "Feltham" = patient.doctor-name
```

To do this in a centralised system is easy, but if in a distributed system the patient relation was distributed over a number of sites, it would become more complex. It would be very inefficient to do in the same way as a centralised system. In a centralised system, each tuple would be retrieved individually. In a distributed system, the tuples would be grouped together and sent as a group and only the attributes required would be sent, not the whole tuple. The query would be sent to all sites and therefore could be done in parallel.

This introduces the whole question of query optimisation. Which is the most efficient, cost effective, and fastest method of performing this query? There has been a lot of research into this area over the last few years as people realize that the methods used in centralised systems are not suitable for use in distributed systems. There are a number of different ways of approaching optimisation. Most researchers have assumed transmission as the target for optimisation. There are also a number of other approaches. The average response time and parallelism achieved in the query are also other areas being considered. However, they are all interconnected in some way and in general a combination of these approaches is usually used in practice. [SEGE86], [ULLM82], [YU84], [MAHM79], [GARC79] and [APER83] all discuss aspects of query optimisation. Query Optimisation is covered in Chapter 5 in more detail.

Integrity, Concurrency Control and Recovery

These three aspects of distributed database while quite different, are strongly interrelated. An atomic transaction is a series of events, that either all happen in entirety, or none happen at all. This ensures a consistent database, and the integrity of the database.

Recovery is related in two ways. The first is when a site either becomes unavailable, or the actual database becomes corrupted. In the first case, the one where most research is being done, there has to be a method of ensuring that the site that is brought back up, and has a database that is consistent with all the others. A transaction could be stopped in mid-stream. In this case should transactions be aborted, or should they be logged until the site is operational again? [EAGE83] talks about achieving robustness in distributed database.

To ensure a consistent database, distributed concurrency control is required. There have been a number of different methods proposed to

handle this problem. There is the 2 phase locking method, which is similar to the method used in single site database management systems. Then there is timestamp based control methods, which use unique transaction identifiers to order transactions. This method is used in SDD-1. There is optimistic concurrency control [SCHL81] and [KUNG81], and the majority consensus method [THOM79].

This is important in the radiology system, as some methods require a large amount of communication to achieve concurrency. In the interconnection of the radiology systems, one of the aims is to reduce the amount of communication. The method of concurrency control used in the SNAPS system is introduced in Chapter 6 in the Section on Concurrency Control.

Chapter 2

Database Snapshots

Existing database management systems allow users to operate only on the current database state. However, there are a number of applications that require or will tolerate access to out of date versions of the database. A snapshot reflects a selected portion of a database as of a particular time, without having to execute the transaction at that time.

Snapshots are read only portions of a database. They are important when an application needs an "as of" state of a database. Most periodic reporting falls into this category. For example, a company might do monthly accounting reports. It would define a snapshot of the database at the end of each month, which it could later be used to process into reports. This would allow other applications to carry on and use and modify the database.

Snapshots have been implemented on a number of centralized systems, but in 1980, system supported database snapshots were proposed by Michel Abida and Bruce Lindsay [ABID80]. This paper was the first to directly address the subject of system supported snapshots.

This paper identifies an number of areas where snapshots are useful, and where further research is needed. In a distributed database environment they can be used to store the result of a complex query that has to be performed frequently. It can be performed once and the results used over and over again with out the cost of having to execute the query. This is particularly important in a distributed database, as the join might involve relations stored at many different sites. They can be used to provide data reference locality by defining locally stored snapshots based on data stored

at other sites. If these snapshots are periodically refreshed they provide approximate read only access to remote data, or avoid the maintenance replicated data. This is the area that I have concentrated on in this thesis.

This paper discussed the semantics of snapshots, their definition, how they can be refreshed, how they are implemented, and then possible extensions to the concept of snapshots.

In another paper [ABID81], Abida discusses the definition of derived data in a system called DEREL. This paper defines a method of defining base relations, views, snapshots, and replicated data. It discussed the idea of a view, snapshot definition producing more than one resulting relation. This is important in the SNAPS system, as this is what I do.

In [LIND86], Lindsay introduces an algorithm for snapshot refresh using the differential approach rather than the delete and re-execute approach to refreshing snapshots.

Two unpublished papers by Cooper [COOP86a] and [COOP86b] deal with the problems of snapshots, and attempts to define ways of categorising snapshots on the basis of their composition. In [COOP86b], the possible implementation of snapshots in MIMER, relational database management system, is discussed. Some of the ideas presented in [COOP86a] are discussed in the following sections.

Categories of Snapshot

One way to categorise the type of snapshot is to consider how they are derived. The source of the relation or relations in the snapshot are two aspects. The number of receiver sites is also a factor. Therefore the types of snapshot can be defined in terms of single (S) or multiple (M) source sites (i.e. the relations that are involved in the snapshot are stored at multiple sites). It can be defined in terms of single or multiple source relations,

irrespective of where they are stored. The last factor is the number of sites that will receive the snapshot [COOP86a].

Based on these factors there are eight different types of snapshot.

1. Single Source Site, Single Source Relation, Single Receiver (SSS). This type of snapshot is highly specific to a single user, and it is questionable whether a snapshot is the most efficient way to satisfy this request.

2. Single Source Site, Single Source Relation, Multiple Receiver (SSM). This is one of the important uses of snapshots. This can be likened to a price list being sent out from head office to other offices.

3. Single Source Site, Multiple Source Relation, Single Receiver (SMS). This is similar to type one, but involves a number of joins and is questionable whether snapshots are appropriate in this case.

4. Single Source Site, Multiple Source Relation, Multiple Receiver (SMM). This is one of the most important uses of snapshots. By performing a complex join at one site, and distributing it to a number of sites, processing costs can be reduced.

5. Multiple Source Sites, Single Source Relation, Single Receiver (MSS). This type of snapshot assumes that the source relation is partitioned over a number of sites, either redundantly or not. This type of snapshot is useful if the snapshot is to be accessed frequently. In the SNAPS system, the global snapshot is this type of snapshot. The local source relation from a number of sites is mapped onto the global snapshot. This will be discussed more in Chapter 4 in the Section on the Global Snapshot.

6. Multiple Source Site, Single Source Relation, Multiple Receiver (MSM). This type makes the same assumption as type five about partitioning. This is an important type of snapshot as it performs a union of a number of fragments of the same relation, and sends the result to a number of sites.

7. Multiple Source Site, Multiple Source Relation, Single Receiver (MMS). A complex join involving a number of sites and relations for a single user. This type of snapshot would be expensive to maintain and would probably require that all relations would have to be sent to the receiver's site. The costs of computing the complex join would be justified if the snapshot was to be used extensively.

8. Multiple Source Site, Multiple Source Relation, Multiple Receiver (MMM). The same complex join is required for each user, and if the snapshot is computed only once, and distributed to each receiver, then substantial processing costs can be saved. Like type four, this type of snapshot would probably have an owner that was responsible for the calculation and distribution of the snapshot.

Distribution of Snapshots

Another issue related to the categories of snapshot is the distribution of the database between the sites. There is a direct connection between this factor and the snapshot type as defined in the previous section [COOP86a].

The types most relevant to the SNAPS system are the MSS, SSM and SMM, and the different types of distribution will be discussed with relevance to these types of snapshot. The resource in the following can be considered to be a MSS snapshot, or the global snapshot.

In the first case (See figure 2.1), the owner retains the resource at a single site, and all queries are handled from there.

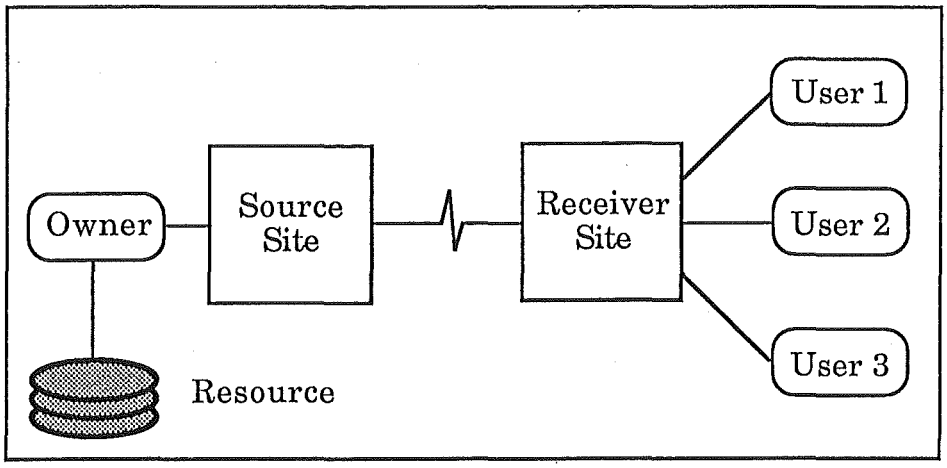

Figure 2.1
Case 1 - Single Resource : Managed by Owner

This is the normal type of processing, where there is only one copy of the database. This is analogous to a centralised database with remote access. This is the situation that we are trying to avoid, as the communication traffic will be high.

This situation is not relevant to the types of snapshot we are dealing with in the radiology system. It is more relevant to SSS and SMS type of snapshot or where all the receivers are at one site.

In the second case (See Figure 2.2), the resource is still managed centrally, but the users at the receiver nodes maintain their own snapshots.

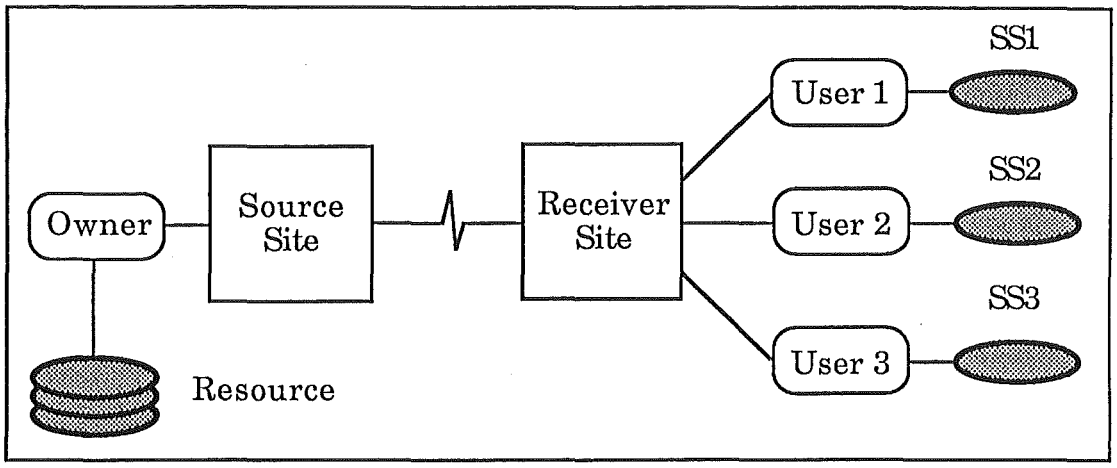

Figure 2.2
Case 2 - Resource Distributed : Snapshots Managed by Users

This situation could produce duplicate snapshots at the same receiver node maintained by different users, which are clearly redundant. It could be useful if the snapshots were significantly different. (e.g. an accounts department, and sales department would access different parts of the database and create different snapshots).

This is relevant to SSM and SMM types of snapshots where the owner wants to exert control over the users and creators of the snapshots.

The third case (See Figure 2.3), is the one that is of most relevance cases as it is similar to how I have distributed snapshots in the SNAPS system,

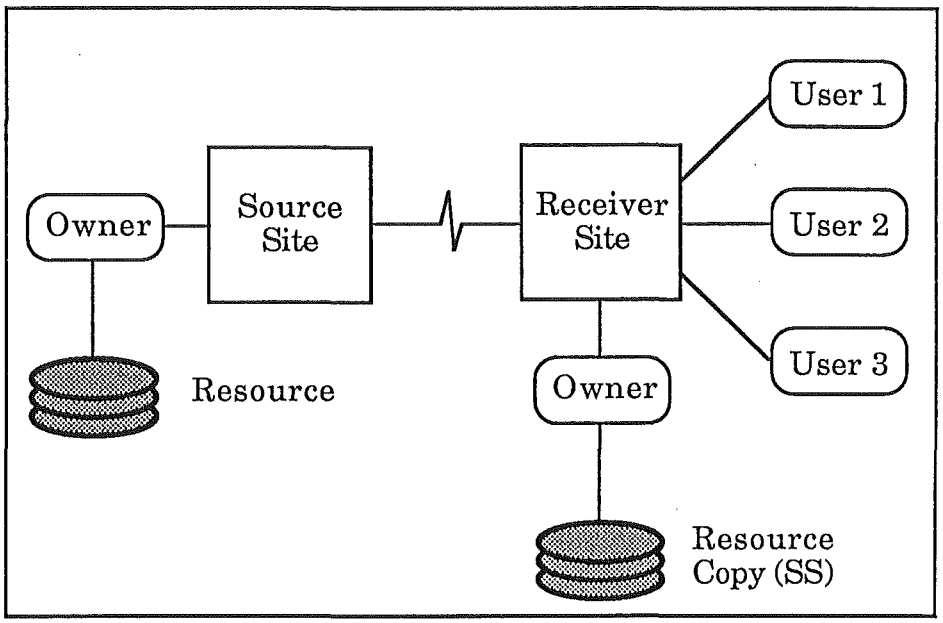

Figure 2.3
Case 3 - Resource Distributed : Snapshot Copy : Both Managed by Owner

In this case the owner maintains control over the central resource, as well as a snapshotted view of the resource at the receiver site. It is the owners responsibility not the users, which is one of the important aspects of the SNAPS system which will be discussed in later chapters. This snapshotted view would be periodically updated by the owner.

This is particularly relevant to the SSM and SMM type of snapshot where the parts of the database that users want to access are distributed, as snapshots to those sites, and queries are satisfied from this snapshot.

It is also relevant to the SSS and SMS, where the ratio of enquiries to updates is high.

In case 4 (See Figure 2.4), the owner maintains both the central resource, and the snapshotted view at the receiver site. There are also snapshots maintained by the users at the receiver sites.

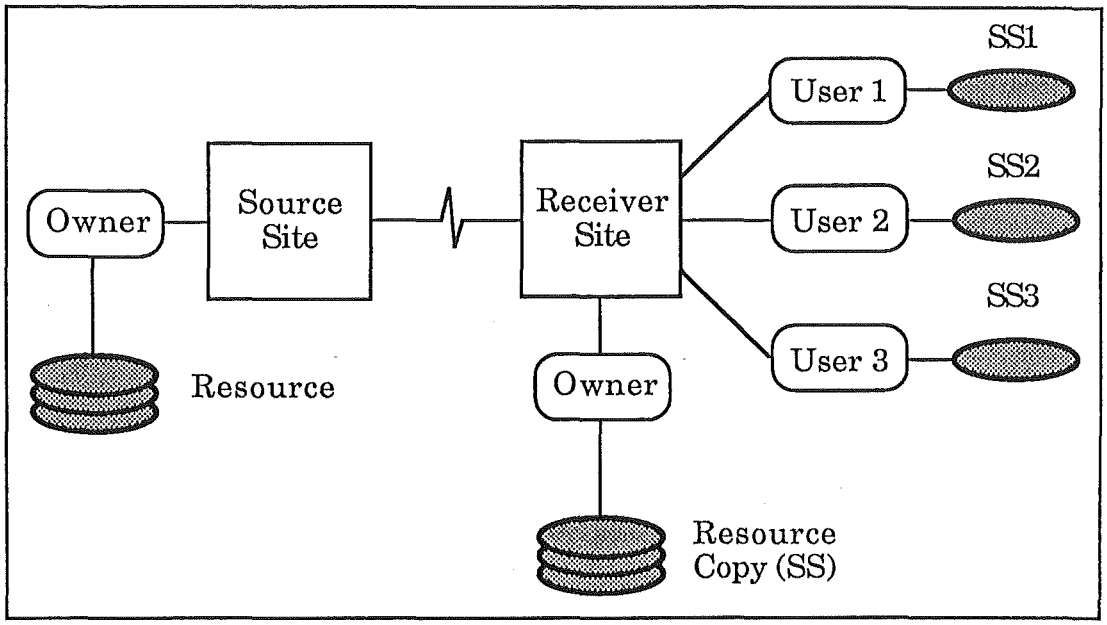

Figure 2.4
Case 4 - Distributed Resource : Snapshot Copy Managed by Owner
Snapshots Managed by Users

As for the second case, if the snapshots were significantly different and the ratio of queries to updates was high then this method could be relevant. Also, the more complex the snapshots, (i.e. type SMM) then the more relevance these two methods have.

The four cases above identify the type of distribution that are possible. The second and third cases are the ones most relevant to the SNAPS system. These will be discussed in chapter 4, which discusses the local databases and the global snapshot, and in chapter 6, which describes how snapshots have been implemented in the SNAPS system.

Snapshot Semantics

Snapshots are stored in separate relations and can be accessed and used like any ordinary relations in the database. Queries can be directed at them and indexes created on them to improve access performance. They can also be used as source relations for other snapshots. This aspect is used in the SNAPS system and is discussed more in Chapter 6. The

relations that contributed to the snapshot continue their own independent evolution. This point will be discussed more in Chapter 6, as it raises some interesting issues with regard to the SNAPS system.

Traditionally snapshots were defined explicitly by running by a program or query executed by a user. This means that the relationship between the snapshot and the rest of the database is embodied in the program that populates the snapshot. In the system proposed by Abida and Lindsay the snapshot would be based on a query on the database, which defines how the snapshot should be populated. This allows the system to automatically populate a snapshot with the tuples resulting from the evaluation of the query at the time of the execution of the snapshot definition command. As the snapshot definition is declared to the database management system the system can refresh the snapshot contents when required, either on demand or after a fixed period of time. The area of snapshot refresh is discussed late in this chapter.

The definition of a snapshot is similar to that of a database view [DATE86]. The difference between a view and a snapshot is that the snapshot is a static copy and a view is a dynamic "window" of the database. Any change in the value of a relation would automatically alter the value of the view. The change would only be reflected in the snapshot, when the snapshot was refreshed. Updates run counter to the intended semantics of snapshots as copies of selected parts of the database made at specific times. In the system I propose for updatable snapshots, this idea is not violated. In [ABID80], it is says that updates to a snapshot should not be propagated to the source relations, because these relations evolve on their own. In the SNAPS system it is proposed that updates to a snapshot are also made to the corresponding source relations if the source relations have not been changed since the snapshot has been calculated. This aspect will be discussed in Chapter 6 in the Section on Updatable Snapshots.

Snapshots are stored in separate relations from those on which they are based. In the SNAPS system, snapshots can be stored in the same relations without causing conflict.

The lifetime of a snapshot starts with the execution of a DEFINE SNAPSHOT command. In a query language such as QUEL [STON76], the snapshot creating statement would look like

```
define snapshot <snapshot-name> (<attribute-list>)
       as <query>
```

The <attribute-list> defines the names of the snapshot attributes. The attribute types are inherited from the attributes of the <query>. The query can be any query that can be expressed in the query language being used. It can include joins, restrictions, projections, and aggregate functions where they are available.

As soon as the snapshot has been defined, it can be used in any query, and take part in the definition of other snapshots or views.

Consider the following simple database consisting of scaled down versions of the local database defined in Appendix H.

```
patient ( pno, name, address, age )
doctor ( code, name, phone )
visit ( pno, code, date )
```

Then the following QUEL statement

```
define snapshot by-jones ( name, date ) as

     retrieve patient.name , visit.date
        where visit.pno    = patient.pno
          and visit.code   = doctor.code
          and doctor.name = "Jones"
```

defines a snapshot containing all the patients that doctor Jones has seen, and on what date their visits were.

Refreshing Snapshots

Snapshots are intended to provide users with a particular state of a database. For example, users might defines snapshots representing "last month's sales", or "today's orders". User control over when the snapshot is refreshed extends the snapshot definition construct. Snapshots can be refreshed on demand with a QUEL command like

```
refresh snapshot <snapshot-name>
```

which will cause the system to replace the current snapshot contents with the results of evaluating the snapshot query in the current database state.

Another approach is to have the system support periodic refresh. This is supported by extending the snapshot definition command, which becomes

```
define snapshot <snapshot-name> (<attribute-list>)
       as <query>
       [refresh every <period>]
```

where

```
<period> ::= year | month | week | day | hour
```

At the end of the refresh period the system refreshes the snapshot from the current database state. Periodic refreshes by the system enhances the semantic utility of the snapshot construct. No only can users specify what parts of the database they want captured by the snapshot, they can also say when new versions of the snapshot should be captured.

The area represents where there are the most interesting implementation problems and solutions. Refreshing snapshots consists of (logically) replacing the current snapshot contents with the results of evaluating the snapshot query in the current database state.

Because the same query must be evaluated when the snapshot is defined and refreshed, some advantage should be taken of the query compilation to

reduce the overhead of query evaluation during snapshot refresh. In some database management systems, such as System R [ASTR76], access modules of queries are stored so that they can be executed without the overhead of query analysis and optimisation.

Because snapshots can always be refreshed from the current state of the database, snapshots do not require the usual database recovery facilities. In the case of a system failure, the snapshots can be re-evaluated. This is acceptable if the relations involved in the snapshot do not change very much. However, a snapshot which is based on relations that changed frequently, some applications may operate on different versions of the snapshot. If two applications used the same snapshot, and one application ran before a system crash. If the snapshot had to be refreshed after the system crash, and the second application ran after the system crash and snapshot refresh, then the two application would have used different versions of the snapshot. This situation could cause problems, and some mechanism would have to be able to get back to the state of the database, possibly using the transaction log, when the last snapshot refresh was performed to reconstruct the snapshot correctly.

The straightforward implementation of refresh would simply be to delete all the tuples from the snapshot relation and then insert the tuples returned by executing the snapshot query. This can be expensive if the number of changes since the last refresh is small compared to the size of the snapshot. It would also be expensive if the the query involved complex joins over a number of sites in a distributed database.

If the snapshot is sufficiently simple, e.g. only based on one relation, then the snapshot could be updated based on the updates on the relation. Any insertions could be checked to see if they satisfy the restriction in the query, and any deletions and updates could be similarly checked. This is

the approach I take in the SNAPS system, as the size of some of the snapshots is quite large, and the number of updates in very small, and the snapshots are generally based on single relations. For example

```
define snapshot patient-50+ ( patient-name, age ) as

        retrieve ( patient.name, patient.age )
           where patient.age >= 50
```

This snapshot could be updated with reference to the changes made to the patient relation, to see if any tuples in the snapshot should be modified. This approach is called differential refresh as opposed to delete and re-execute refresh.

Aggregate functions, such as MAX, MIN, AVG, SUM and COUNT would require more complex processing but can be done. For example

```
define snapshot seen-jones ( doctor, total-patients ) as

        retrieve (visit.gp, total = (count(visit.pno)))
           where visit.gp   = "Jones"
             and visit.date = "12/5/63"
```

shows an example of a aggregate function in a snapshot that could be updated by applying the modifications to the **visit** relation to the snapshot.

For snapshots that involve more than one source relation, the updating problem becomes more complex. Joins especially over a number of sites in a distributed database, would require large amounts of communication and processing to perform differential refresh, and it would be easier to delete the snapshot and re-calculate the snapshot. This problem is related to the view update problem [BANC80], and like the view update will probably require the re-evaluation of the snapshot query.

Extensions to Snapshots

In [ABID80], a number of possible extensions are discussed. Increased user control over the refresh period would give the user more flexible

control over the contents of the snapshot (e.g. every 4 hours). More control over the base time (e.g. every week on Tuesday at 3am) would allow for flexible scheduling when the system might be lightly used, or utilise lower processing and/or communication charges.

In [COOP86a], the notion of a snapview is introduced. It is proposed that a snapshot would be controlled by the owner of the relations being responsible for the calculation and distribution of the snapshot. This is like a head office sending out a price list. A snapview would be a user requested snapshot, maintained by the user not the owner. This would be like a radiologist requesting details of a new patient. This aspect is important, and will be discussed in Chapter 4, with reference to the SNAPS system.

The concept of a future update is proposed. This mechanism would be used to have updates performed on snapshots at a predetermined time in the future. The updates could be logged and then performed against the snapshots at a latter date. This type of mechanism is used to define the system of updatable snapshot in Chapter 6.

Chapter 3

The radiology system was first proposed by Dr Kennedy, Head of the X-ray department at Christchurch Public Hospital, and Dr Feltham, Head of the X-ray Department at Nelson Hospital in late 1985 [KENN85] and [FELT85]. The initial design of the system was done by David Tripp and Alaistair Kenworthy, Honours students in Computer Science at the University of Canterbury in late 1985 and early 1986 [TRIP86]. The development was to be done in conjunction with Concept Data Systems, a software development house based in Christchurch,[CONC86], and the Christchurch Public Hospital.

Initially the system was to be installed in Christchurch in St Georges Hospital, Christchurch Public Hospital and the Gloucester Radiology Clinic. It was also being developed for Nelson Hospital and the Bridge Street Radiology Clinic. The first part of this chapter discusses the functions of the radiology system.

In discussions with the above doctors and Dr R.E.M. Cooper, the then Head of the Computer Science Department at the University of Canterbury, and myself, the possibility of interconnecting the systems at the individual hospitals and clinics was discussed. The second part of this chapter presents the requirements of the interconnection of the radiology systems.

There were a number of options to be considered for the interconnection of the sites in the two systems that would be compatible with the G/Net local area networks. A study of the requirements for the wide area network for both the Christchurch and Nelson systems was done by myself, and is presented in Appendix E, and recommendations for both these systems are presented later in this chapter.

The Radiology System

A feasibility study done by Tripp and Kenworthy [TRIP87], identified the following areas and the functions within these areas, that were to be handled by the radiology system.

- patients
    - maintain patient details
- visits
    - schedule appointments
    - record type of x-rays
    - create visit labels
    - record medical supervisors
    - record reports
    - organise distribution of reports to other doctors
- accounting
    - calculates charges for the x-rays
    - prepare invoice to patients
    - receive payments
    - calculates the ACC[1] claim and health department subsidy
    - perform audits
- send messages between staff members
- handle requests for x-rays from other sites
- provide statistics on types of examinations, radiologists,patient type, and room utilisation
- research facilities

At each site there is a network of IBM personal computers or IBM compatible computers linked using the local area network(LAN),

[1] ACC is an acronym for the Accident Compensation Corporation

G/Net.[GATE87]. Each site has a small file server with between 40 and 80 megabytes capacity.

The application is written in a combination of Informix [INFO86], and using its C [KERN78], language interface. The program used the relational database management system within Informix which has a SQL interface [DATE86]. A local database was developed and the schema is shown in Appendix H.

Interconnection Requirements

To increase the power and effectiveness of the system the radiologist wanted to have access to patient details and reports of x-rays performed not only at his clinic or hospital, but also those done at other hospitals or clinics [KENN85]. This was particularly important for the Accident and Emergency Department at Christchurch Hospital. It is generally not possible to get hold of a patient's record or copies of previous x-rays in the case of an emergency. The doctor needs to be able to have access to this information immediately.

The system was to provide security at various levels for sensitive medical information. Because the system was to interconnect public and private organisations there had to be security not only for the data, but also for access to the other systems. The security of the data was handled by the single site system, by restricting users to various levels of security from the receptionist who could access only those parts of the database he/she needed, to the radiologist who has the ability to access to all parts of the database. There were various level in between these. This is done by only allowing access to certain relations in the database. The Informix [INFO86], system performs automatic checks on the user code against those that have read, or write or no access to a relation each time a relation is accessed.

It was decided to have only selected parts of the local database available to the other sites. This was done as some of the information that was stored in the same relations was not unique to one site. (e.g. a patient can hold an account at more than one site, which will contain different information). It was done as some of the information stored is not relevant to other sites. (e.g. time of x-ray appointment in visit relation). Some of the information is also commercially sensitive. (e.g. accounts of the patient, clinics, and hospitals).

In discussion with Dr Kennedy and myself the data that was to be made available to the other sites was decided on. This is defined in the schema of the global snapshot as shown in Appendix I, and is based on the local schema in Appendix H.

The decision to map the local databases on to a global snapshot was partially based on the requirement that not all data was to be made available to the other sites. The concept of the global snapshot and the mapping onto a local database is described in Chapter 4.

The radiology database in predominantly read only and the ratio of enquiries to updates would be very high. There are some parts of the database where it would be useful for users at other sites to be able to update. This data would be stored as part of the global snapshot. This requirement was the basis of providing updatable snapshots in the system.

One of the radiology system requirements is that the local sites have a high level of site autonomy. This implies that each local site has very little control over what happens at another local site. This is important because of the type of information that is being stored by the system. It is also important because the organisations running the systems are public organisations, and private companies or hospitals, and they are independently run. The system is based on micro computers linked

together on a local area network. There is therefore no dedicated computer that could act as a controller for a distributed database. This means that somewhere in the system there has to be a controller to co-ordinate all the sites. This was the basis of the decision to have a central controller which provided restricted access to the local databases and managed the updates of data stored at other sites.

One of the problems with the interconnection of the radiology systems is that there has to be unique keys in a number of relations that are going to be accessed by other sites, particularly in the patient and visit relations. This problem is discussed in [MUNZ79] with reference to the VDN distributed database management system. To achieve some sort of uniqueness, a site code should be added to the keys of relations that are mapped onto the global snapshot, that will identify each tuple uniquely. This raises problems if the global snapshot site becomes unavailable. Then the global snapshot can not be checked to see if a particular record exists, and then a new conflicting record could be created. There would have to be some manual system that could be used to check if duplicate records with different key values had been added to the local database when the global snapshot was not available.

Wide Area Network Requirements

The local area networks used by all the site was the G/Net LAN system. To provide communications between the sites there had to be a flexible, easy to use and expandable type of wide area network. It was decided that the product G/X25 [GATE85], was the best possible option. A full description of the G/X25 product can be found in [GATE85] and a summary in Appendix F. This product provided a gateway for users on a G/Net LAN to a X.25 Public Data Network PDN, and provided remote invisible file access, one of the major requirements of the network system.

A study was made of the three possible options that could be used with the G/X25 system for the wide area network. The only supplier of wide area networks in New Zealand is Telecom. The options were leased lines, the X.25 packet switching network PacNet, and the digital data network, DDN.

The study considered each option with the following objectives in mind:

- Minimize installation, and usage costs
- Ease of implementation
- Flexibility of operation, and expansion

This study is presented in Appendix E and in the following sections the recommendations for Christchurch and Nelson are made.

Recommendations for the Christchurch System

From the study in appendix E the best option for the Christchurch network is the X.25 packet switching network PacNet.

The leased line option was discounted as the installation costs were very expensive. The requirements to add another site to the network are expensive. It requires two additional G/X25 boards which are expensive and requires another leased line. As the number of sites increase the number of boards into the site where the global snapshot increase. This situation is not very desirable, as it requires that the PCs are always turned on, and this can not be guaranteed in an environment where there is little or no control over users.

The digital data network is similar to the PacNet option but is more expensive in all respect to the PacNet option, and therefore is not considered further.

The PacNet option (See Figure 3.1) allows for easy operation and easy expansion. It is more expensive to operate than the next best option, leased lines, but these other advantages out weigh this. When a new site is added to the network, no other site is affected. The costs of installation are significantly less than that of leased line, and require no modification to the network, or the site of the global snapshot.

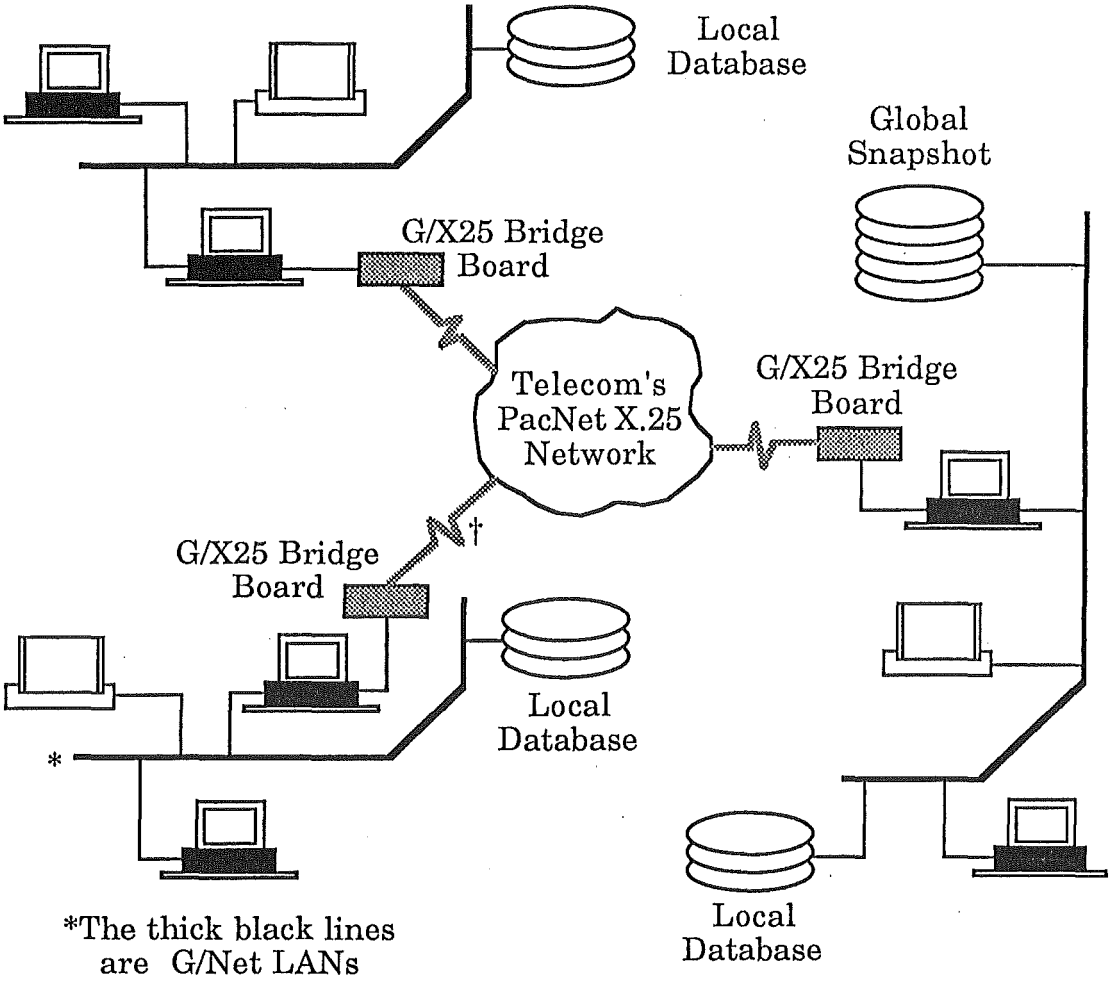

Figure 3.1
Proposed Configuration of the Christchurch Network

The advantages favour the packet switching network as the number of sites increase, and as the number of sites is expected to include all the major hospitals and more private practices, this option would be the preferred one in the long term.

The PacNet option also has a number of other advantages. Security can be provided by the network in terms of a closed user group which restricts access to those sites in the group. These is also capacity in the X.25 definition to encrypt the data.

To reduce the costs of the network the global snapshot should be stored on one of the local networks. In the case of Christchurch, the proposed site is Christchurch Public Hospital. This decision was made as it will likely to be the biggest user of the system, and that if there are problems with the PacNet network this site will not be affected and would still be able to access the global snapshot. This is important as Christchurch Public Hospital is the site of the Accident and Emergency Department for Christchurch.

Recommendations for the Nelson System

The options for the Nelson System were the same for the Christchurch system. However, there were a number of differences between the two systems. The Nelson system has only two sites, Nelson Hospital, and the Bridge Street surgery. There is no likelihood that there will be any other sites. Therefore there was no need to worry about easy of expansion as there was in the Christchurch situation.

The amount of work being done at the Hospital is significantly more than that at the surgery. It was therefore decided that there was no need for a global snapshot and that remote access to the database at the Hospital would be adequate. The relations in the database at the Hospital that would be accessed at the surgery, are the same relations that are mapped in the global snapshot defined in Appendix I. These would need to be modified to remove attributes from these relations and placed in new relations that were required at both sites. This is the same as the problem for the Christchurch system that is alleviated by the mapping system

defined in Chapter 4. The rest of the local database would still be stored at the surgery.

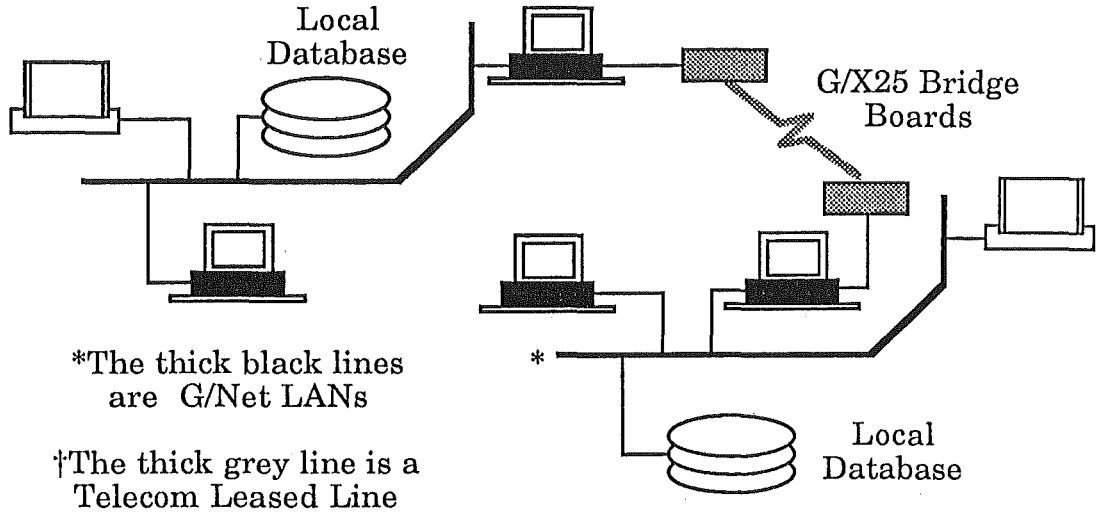

Figure 3.2
Proposed Configuration of the Nelson Network

The network topology decided on is one of a leased line between the two sites which can be used to run the G/Net X.25 system (See Figure 3.2). While the initial costs of installation are expensive the operating costs are cheaper than for PacNet. It offers the same if not better performance than PacNet, as there will be not competition of use of the leased line as there could be with PacNet.

Chapter 4

The Approach to System Interconnection

One of the initial design decisions that had to made was how sophisticated the system was to be to interconnect the different sites. A full distributed database management system offered more power than was need for this system. A distributed database management system is generally designed to work on a number of databases of the same organisation and where there is a local database administrator at each site. There is also generally a large powerful computer at each site to handle incoming requests from other sites.

The requirements of the radiology system have led to the decision to implement a central controller to co-ordinate all the local sites. The functions of this central controller are to provide read only access to selected data stored at the local sites to the other sites. It is required to allow update access to selected data at the local sites. As there is no dedicated computer to handle the database operations at the local sites, the central controller has to take over some of the processing when other sites want information at this site.

The requirements to minimize costs especially communication costs and to have fast access, has meant that some compromises have had to be have been made as to what functions are performed and where.

It was decided that the central controller should also be used to store data. It will function like the storage site which will hold the selected parts of the local databases in a global snapshot, that are to be accessible and updatable at the other local sites. It is assumed that the global snapshot is

based on the schemata of all the local databases, decided by all the local database administrators.

This will fulfil a large number of the requirements. Firstly, there will be a high level of site autonomy as the only site a local site will know about is the central controller's site. This reduces the complexity of performing updates and queries on the database, as access is required only to the site of the global snapshot. This method provides a high level of security as only a pre-defined view of the database is provided, and sites can only access information stored at other sites that the other sites want other users to be able to see.

The disadvantages of using this approach are that the central controller is susceptible to crashes. If the network becomes unavailable or the central controller crashes, then the local sites have no way of communicating with each other. This argument is becoming less of an issue as the reliability of communication networks increases, and the other features included in the system are taken into account.

Another disadvantage is that the central controller could get overloaded. To alleviate this a number of mechanisms have been included into the system to reduce the amount of communications and co-ordination between the central controller and the local databases.

The mechanism for performing an enquiry on the global snapshot checks to see if an enquiry has already been done for this local query. If it has it knows that the data will be up to date as of the previous global snapshot refresh, and an enquiry on the global snapshot will not be required. This mechanism is defined in more detail in the Chapter 5 on Query Optimisation.

If a lock on a tuple has been acquired from the global snapshot, the site does not need to request the lock each time it wants to update the data. It retains the update rights for this tuple until the next global snapshot refresh. This aspect is discused more in Chapter 6 in the Sections on Updatable Snapshots and Concurrency Control.

Another disadvantage of the global snapshot approach is that there is duplication of the database. However as the number of sites increase, the complexity of the system does not, and the overall size of the global snapshot will increase less when a new site is added to the network. When another site is added, no other local sites will be affected.

The major argument against duplication is maintenance of the multiple copies. In the SNAPS system the maintenance costs have been reduced by allowing only one site update rights to a tuple for the period between global snapshot refreshes. The commitment of updates from a site are all done at one time when the global snapshot refresh is done. After that the all the local database snapshots are bought up to date by the global snapshot controller which performs the global snapshot refresh and refreshes the local database snapshots.

The other argument against duplication is that the physical requirements increase. This argument is becoming less important as the monetary costs of storage are coming down all the time. The size of the global snapshot will increase all the time. In the radiology environment, there could be an expiry date when data is no longer stored in the global snapshot. A mechanism has been implemented in the SNAPS system that does this. If a local snapshot is not accessed for a set time period it is deleted from the local database to reduce the build up of snapshots in the local databases.

The advantage of the global snapshot approach is that the complexity of the system is reduced. Queries can be handled much more easily as there is only the site of the global snapshot that has to be accessed. This reduces communications. The system takes advantage of lower communication charges by performing the global snapshot refresh when charges are lower. The system provides a high degree of security and site autonomy to the local sites and their databases. These and other advantages of the system will be introduced in this and later chapters.

Another decision that had to be made was the degree of granularity of locks on the local database and the global snapshot. Granularity is the size of a lockable object in the database. The extremes of this are where the is one lock for the whole database that locks the whole database, and where every value of every attribute in the database has a lock. There has to be some compromise. A low degree of granularity that reduces the storage requirement and decrease the management of the locks, will increase the number of transactions that have to be aborted because the data required has already been locked. A too high degree will require large amounts of storage, increase the cost of maintaining all the locks, and may provide a degree of granularity that is not required by the users.

It was decided for the radiology system that the granularity should be at the tuple level. This is because generally when a user performs an access a single tuple is selected. In the application program for the radiology system, a screen layout generally resulted in one tuple being selected from a relation. It was decided that to achieve as few update transaction aborted because the lock(s) had already been acquired, the degree of granularity should be high. Update locks were going to be acquired for the period of the global snapshot refresh which in the radiology system was each day. Only when the global snapshot refresh is done and updates committed are the locks released.

There are two levels of snapshot in this system. The first level is a global snapshot of all the local databases. This is stored separately from the local databases in a separate database at a completely separate site or on the local area network of one of the local databases. The global snapshot is a MSS type snapshot as defined in Chapter 2. This type of snapshot is a union of a number of fragments of a relation into one relation. In the SNAPS system this has been extended to allow union of one relation, to be mapped onto a number of relations that are vertical partitions of the single relation, with a common key for each of the vertical partition relations.

This chapter discusses how the global snapshot is mapped onto the local database, and vica versa. There are a number of potential problems in this mapping system, that are identified and possible solutions suggested.

The second level is a snapshot of the global snapshot that is requested by a user. This is what was termed a snapview in Chapter 2 : a user requested snapshot. Snapshots in the SNAPS system will be discussed in Chapter 6.

The Global Snapshot

To provide access to selected parts of the local databases to the other sites, it was decided to implement a global snapshot of selected parts of all the local databases. This would be stored separately from the local database in a separate database.

The schema of the global snapshot would be defined by the local database administrators of all the sites involved. From this the **mapping** and **mapping-qualification** relations can be defined, which define the local database to global snapshot mappings. These will be discussed more later in the chapter.

The schema of the global schema must be known to the local database administrators. This is because the keys of the global snapshot relations

must match those of the local database relation that maps onto them. Any attributes stored in the local relation can be mapped onto attributes in the relation(s) in the global snapshot. Not all sites need to know about all the attributes stored in the global snapshot. The mapping relations only know about the attributes of the local database that are mapped onto the global snapshot for this site.

The SNAPS system is based on case two described in Chapter 2 in the Distribution of Resources Section. In that case the resource was stored a single site, and the users requested snapshots from that site. In the SNAPS system, the resource is the global snapshot. From a different perspective the system is also like case three. This example had a snapshot of the resource at the receiver's site from which the system took snapshots. In the SNAPS system I propose that there will be a global snapshot stored at one site from which snapshots will be requested from the local sites. These snapshots will be stored at the local sites, and all users will be able to access them.

The system I propose (See Figure 4.1), is a cross between case two and three, but not the same as case four as no user will have control over the snapshots it has requested. They will still be maintained by the central controller, and be available to all uses at that site. They are also all stored together so that there will be no duplication of snapshots even if parts of them overlap. This was one of the problems with cases two and four. This point will be discussed more later in the chapter.

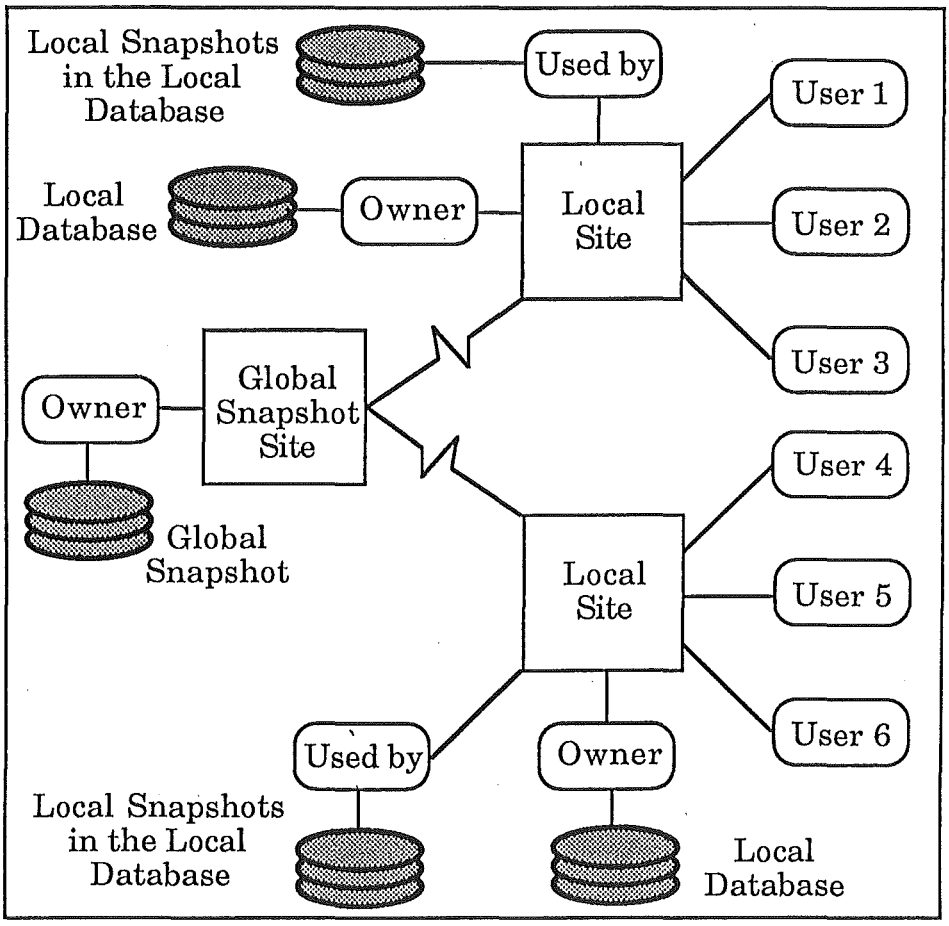

Figure 4.1
Distribution of the Local Databases

Figure 4.1 show the distribution of the global snapshot, and the local databases. It also shows that a sites can not access the local database at other site. All requests have to go through the central controller at the site of the global snapshot. It shows that the user snapshots are maintained by the central controller site. The local database that is not mapped onto the global snapshot is still maintained by the local database administrators.

Any relation stored in the global snapshot has an extra attribute added to it which has a number of status variables within this extra attribute, that are used to maintain the global snapshot. Figure 4.2 shows the layout of the attribute, and in Appendix B a full description of the attribute can be found.

This attribute is used to maintain the local snapshots, by storing information about which sites have a copy of this tuple, which site has requested and received an update lock on this tuple.

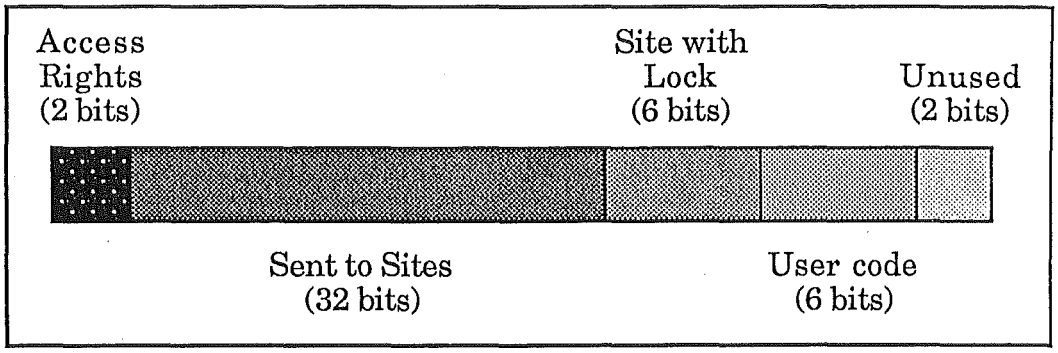

Figure 4.2
Format of the Global Snapshot Relation Status Attribute

This attribute contains the access rights to the tuple. This indicates the state of this tuple. The rights can be locked, which indicates that a site has request a lock on this tuple. It can be read/write which means that no site has acquired the update rights to this tuple. The read/only state indicates that this relation is read/only and no updates can be performed on this or any other tuple in the relation. The state can also be temporary, which indicates that the tuple is not in the relation permanently, and is being used by a join operation on this relation. As its name indicates, the tuple will be deleted when the the join operation is finished. This variable will be discussed in Chapter 5 on Query Optimisation, in Chapter 6 in the Section on Concurrency Control, and in Chapter 7 on Algorithms, its use will be shown.

To restrict those relations in the global snapshot that can be updated, some relations are only read only, and some are read/write. This information is stored in the access rights flag in the status attribute. If the flag is in the read only state then no site can update the data apart from the owner. If the state is read/write, then any user at any site that is allowed access to this relation is allowed to update this relation.

A bit map of the sites that this tuple has been sent to as part of a snapshot is stored. Up to 32 sites have been allowed for. This variable is used so that any changes to this tuple can be propagated to the local snapshots that have a copy of it as part of a local snapshot. It is also used to check to see if a site already has a copy of this tuple in its local database, so that another enquiry on the global snapshot does not have to copy this tuple again. This variable will be discussed more in chapter 6 on how snapshots have implemented in the SNAPS system, and the use shown in Chapter 7 on Algorithms.

The code of the site that has placed a lock on this tuple is stored. This is done so that when the global snapshot refresh is done, the central controller can get the changes made to this tuple, and then send it to the other sites. If the access rights for this relation are read only then this variable indicates the owner of the tuple. This is used in the global snapshot refresh and its uses will be shown in Chapter 7 on Algorithms.

The code of the user that has accessed this tuple is also stored. This is used when more than one user at the same site is accessing this relation, and the access rights variable is set to temporary. This distinguishes the users at the same site when two or more users are performing joins on this table. This will be discussed more in Chapter 5 on Query Optimisation, and in Chapter 7 on Algorithms.

The Local Database

The requirement that there be a high degree of site autonomy has meant that the local database at each site is independent of local databases at other sites. Having the global snapshot in between the the local databases provides this autonomy. The central controller controls all access to data in a local database by having a global snapshot of the data that site is

allowing others to use, and no other site has direct access to another sites database.

There is some loss of autonomy, when a relation in the global snapshot can be updated. This means that data stored in a local database and that is mapped into the global snapshot can be updated by another site without the permission of the owner of the data. However the data stored in a relation that can be updated is the type of data that all users at all sites are allowed to update, as it is not generally sensitive data, and it is not specific to one site. (e.g. a patient's detail like name and address). If it can be updated then all sites would have had to agree that other sites can update this information without the owners knowledge, when the schema of the global snapshot was being defined.

For those relations in the global snapshot that are read only by all sites, then the owner is the only user that can update the data. This is the case for most of the database. (e.g. a visit report would be read only as it is sensitive data).

As for the global snapshot, all relations that are stored in the local database, and are going to be made available to the other sites must have an extra attribute added. The lay out of the status attribute is shown in Figure 4.3, and a detailed description can be found in Appendix A.

This attribute is one byte long, and stores the following information : the tuple type, the current access rights of this site for this tuple, and flags indicating if the data is new, or has been modified since the last global snapshot refresh.

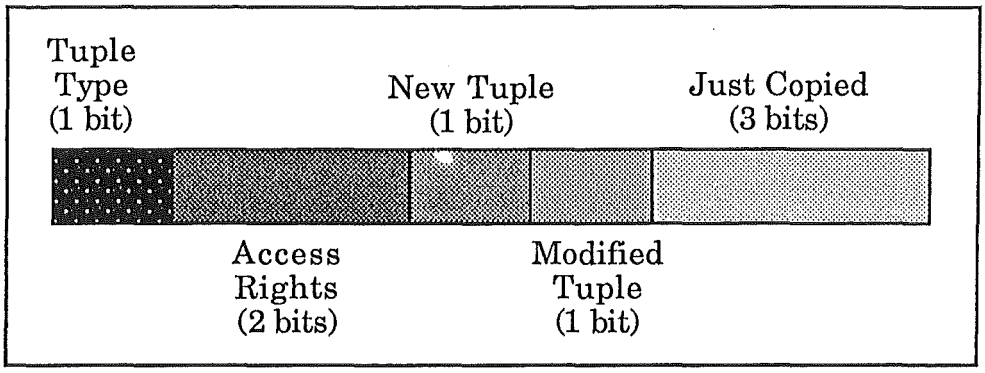

Figure 4.3
Diagram of the Local Database Relation Status Attribute

The tuple type indicates if the tuple is part of a local snapshot or if this site is the owner of the tuple. If the tuple is part of a snapshot, then the tuple identifier of this tuple and the snapshot number and the local relation name will be stored in the **snapshot-description** relation. This snapshot number is the key of another relation, the **snapshot-definition** relation, which stores target-list and the query qualification of the snapshot. The implementation of snapshots in the SNAPS system will be defined in more detail in Chapter 6, and the use of the mapping relation is discussed in Chapter 5 on Query Optimisation, and in Chapter 7 on Algorithms.

The current access rights indicates which state the tuple is in. It can be in either of the four different states read/write, read only, unknown and deleted.

The read/write state indicates that this site has update access on this tuple. This indicates either that this site has acquired the access rights from the central controller. This site retains the update rights for this tuple until the next global snapshot refresh.

If the flag is unknown then the relation can be updated, but no user at this site has attempted to update this tuple.

If the state is read only then only those tuples which this site is the owner of can be updated.

If the state is deleted then this acts as a log, so when the global snapshot refresh is performed, the corresponding tuple(s) can also be deleted from the global snapshot.

The new tuple flag indicates that this tuple has been added by this site to the local database since the last global snapshot refresh. This is used when the global snapshot is being refreshed. The global snapshot refresh performs a query on each local relation that is mapped to the global snapshot, on this flag.

It is also used when a query involves a join is needed on two relations. As the global snapshot contains all tuples from this site in it already, all that is needed to perform the join for this site are the tuples that have been added since the last global snapshot refresh. This use is discussed in more detail in Chapter 5 on Query Optimisation, and in Chapter 7 on Algorithms.

The modified tuple flag is only used for tuples which this site is the owner of. If this tuple is part of a snapshot, then the access rights will indicate if the tuple have been modified. However if the tuple is a local one, and the relation is a read only one, then there is no way of identifying if a tuple has been modified by this method. It is used when the global snapshot refresh is being done, and also when a query is being done, similar to the method used for the the new tuple flag.

Mapping between the Local Database and Global Snapshot

The key to the whole system is the mapping of the local database schema onto the global snapshot and vica versa. This defines the how the global

snapshot is defined, and what attributes are mapped onto the local snapshots when an enquiry is done on the global snapshot.

The mapping system was modeled in the Ingres relational database management system. The local databases and global snapshot were modeled in the same database with the mapping relations providing the connection.

The mapping system can handle one to many mapping : one local relation can be mapped onto many relations in the global snapshot. This approach allows for any possible mapping of the local database onto the global snapshot. It is assumed that an attribute is not stored redundantly in the global snapshot, and if an attribute is stored redundantly in the local database attribute it is only mapped once onto the global snapshot. This does not apply to the keys of relations which are the same in a number of relations. This removes the possibility of having to maintain a many to many mapping, which would case consistency problems for both the local database and the global snapshot. If the attribute is stored redundantly in the local database, then it is assumed that there are mechanisms in the user application accessing the database to maintain the redundant copy. This is the case in a normal centralised database system.

It is assumed that if there is a one to many mapping that all the participating relations in the global snapshot have the same key. This is to ensure that there is a unique way of identifying and remapping the tuple in the global snapshot back onto the local database relation. Later in this chapter, a method is suggested that can handle a local relation that has a non-unique key and is mapped onto multiple relations.

The mapping between local database and the global snapshot is maintained by two relations : the **mapping-description** relation and the **mapping-qualification** relation. The detailed definitions of these can be

found in Appendix C and Appendix D respectively. Each site has its own set of mapping relations, and are independent of the mapping relations at other sites. This allows different sites to have different mappings onto the global schema. This is an important example of site autonomy as each site is free to modify its own schema without reference to other sites.

The first relation is the **mapping** relation, which has the following attributes : **local-relation, local-attribute, global-relation**, and **global-attribute**. There is a **mapping** relation for each site, and it will be different for each site.

The second relation is the **mapping-qualification** relation which stores the extra qualifications, if there is a one to many mapping, and has the attributes **local-relation** and **qualification-loc-to-cent.**

The **local-relation** and **local-attribute** fields identify an attribute and relation in the local database, and the **global-relation** and **global-attribute** fields identify the attribute and relation that it is mapped onto in the global snapshot. The **qualification-loc-to-cent** field stores the qualification, that has to be added to the query qualification when mapping the local database relation onto the global snapshot. There is no facility for type conversion, as it is assumed that the types of attributes in the global snapshot is the same as those in the local databases. This assumption was made for the radiology system as the initial systems were all the same, and had the same database schema. A possible extension to this is discussed later in the chapter.

When a mapping is being performed from the local database to the global snapshot, a query is performed on the **local-relation** attribute of the **mapping** relation. For a mapping back the other way, a query is performed on the **global-relation** attribute. These queries return the target

lists of the queries that are performed on the queries on the local database relation of one the global snapshot.

Figure 4.4 shows an example how the visit relation in the local databases is mapped onto one relation in the global snapshot.

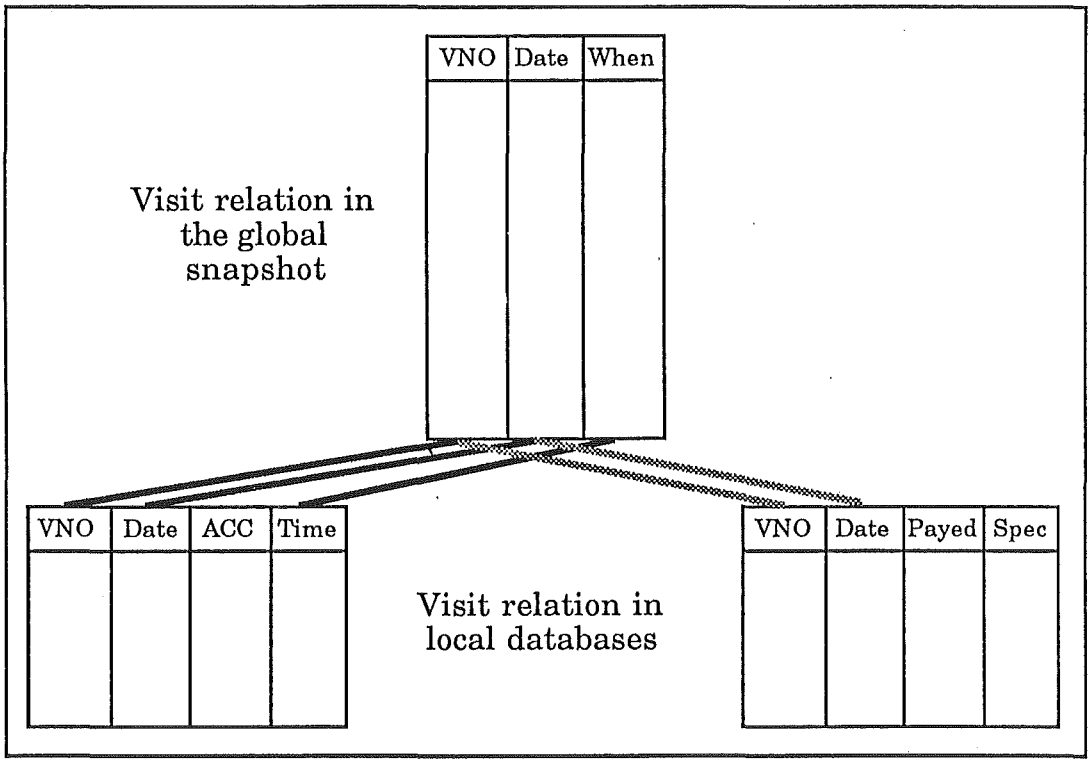

Figure 4.4
Diagram of mapping of two local database relations at different sites onto a single global snapshot relation

Figure 4.4 shows the local visit relations at different sites, and how they map onto the global snapshot. The first local relation has mapped three of its attributes onto the three attributes in the global snapshot. The second local relation has mapped only two of its attributes onto the global relation. Tuples from the second relation will have a blank or null value in the global snapshot for the missing attribute.

Figure 4.4 shows that not all sites have exactly the same map of the global snapshot. This is done as some sites might not want access, or might be restricted for security reasons, to all the data in a relation. It also shows

that the local relations do not have to be identical. In this case there are some attributes that are different in the local relations.

One of the problems that has been identified in that of when the mapping at two sites are different. In the case above, when a snapshot of the visit relation is created for the first site, there will be a number tuples from the second site that will have the **time** attribute set to null or zero, as this attribute is not mapped onto the global snapshot. What happens if the first site is able to update the **time** attribute. There would be an inconsistency in the database. However nobody would notice the inconsistency, as nobody can access the second site, and the global snapshot controller does not know if the **time** attribute is even stored in the second database. Therefore this inconsistency is unknown to any user so the problem can be ignored.

A more serious problem occurs when an attribute in a tuple that is not part of the mapping onto the global snapshot, and the tuple is part of a local snapshot, is updated. The value of this attribute would have been null or zero when the tuple was loaded into the local database. In the case above, the **ACC** attribute was updated in a tuple that was part of a local snapshot. This effectively means that this tuple should become a totally new tuple that is owned by this site. This situation is handled by setting the number-of-days-since-last-access variable in the snapshot-definition relation to the permanent state. This way the tuple will be permanently in the local database until it is removed by another operation. Any changes to the attributes of the tuple that are part of the local snapshot will still be propagated to this site. This tuple can not be removed by the mechanism that deletes unused snapshots after a fixed period of time, and is also not deleted during local snapshot refresh.

Figure 4.4 also shows the attribute name in the local databases does not need to be the same as the attribute it maps onto in the global snapshot

(e.g. the **time** attribute in the first local relation is mapped onto the attribute **when** in the global snapshot).

The mapping relation for the first local database would look like this.

local relation	local attribute	global relation	global attribute
visit	vno	visit	vno
visit	date	visit	date
visit	time	visit	when

In this case there are three attributes in the visit relation mapped onto the global snapshot with the first two having the same names for both relations and attributes, and the last attribute **time**, being mapped on an attribute called **when**.

The facility to allow for one local relation to be mapped onto more than one relation in the global snapshot is done so that it is possible to reduce the amount of access and resulting I/O when a query of the global snapshot is done. It could be useful to use this feature if a site only mapped part of the local relation and another site mapped the other part of the same local relation. If a number of sites want to map part of the global snapshot, then the schema of the local database should be changed to reflect the schema of the global snapshot more closely. Any changes to the schema of the global snapshot would have to be checked against all the mappings of the local databases to ensure that the changes did not affect their mapping system. Any changes made to the mapping of a local database effect only that site, and therefore can bd done without the other sites knowing about the changes for that site.

Figure 4.5 shows the visit relation in the local database has been split into two relations in the global snapshot.

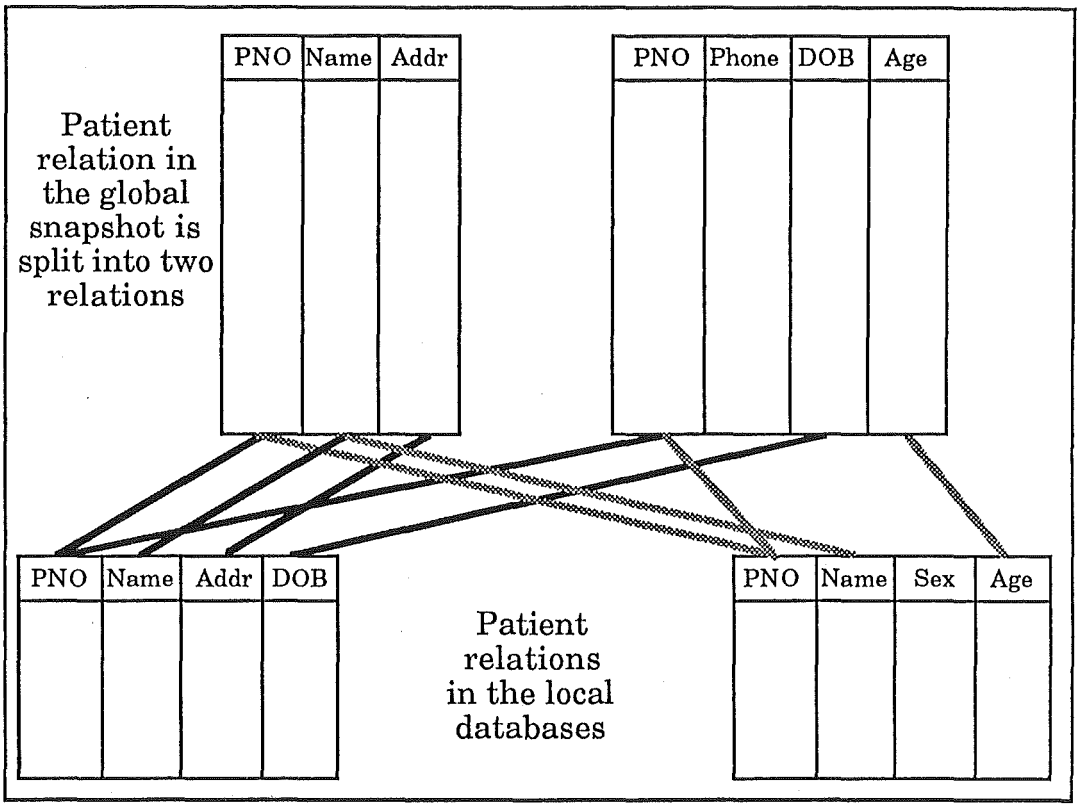

Figure 4.5
Diagram of two local relation at different sites mapping onto two global snapshot relations with a unique key of the local relations and the global snapshot being the attribute pno

To make this system more efficient, the key of the local relation, in this example the **pno** attribute, is stored in both of the global snapshot relations. This done as the most likely type of query will involve the access on the key. This does create problems when a site is mapped onto attributes stored in both of the global snapshot relations. This problem is solved by adding a join operation on the key of the global snapshot relations involved in the map. This mechanism is effective if the key is unique. It uses the **qualification-loc-to-cent** field in the **mapping-qualification** relation as defined before, which is added to the query.

For example, a query on the local database patient relation with the qualification "name = Jones" :

```
retrieve patient.all
   where patient.name = "Jones"
```

would be mapped onto the query with the mappings as those indicated with the lines in Figure 4.5 and the names of the two global snapshot relations being **pat-addr** and **pat-other** respectively and the **qualification-loc-to-cent** attribute being "pat-addr.pno = pat-other.pno".

```
retrieve pat-addr.pno, pat-addr.name, pat-addr.addr,
         pat-other.dob
   where pat-addr.name = "Jones"
     and pat-addr.pno  = pat-other.pno
```

This example show the use of the **qualification-loc-to-cent** attribute from the **mapping-qualification** relation so that the correct tuple is mapped in the **pat-other** relation.

However, in the case where the key is not unique, then two tuples in the local database may have the same key, but their other attributes may differ. In this case there has to be some other way to map the two global snapshot relations correctly onto the local database.

This is achieved by making one of the relations in the mapping, the main relation. The unique tuple identifier of this relation is stored in the other relations in the map as an extra attribute. This is based on the assumption made earlier, that there is a unique tuple identifier for each tuple in each relation.

For example, the visit relation in the first local database in Figure 4.6 is mapped onto two relations in the global snapshot, one with name and phone, and the other name and date of birth amongst other attributes.

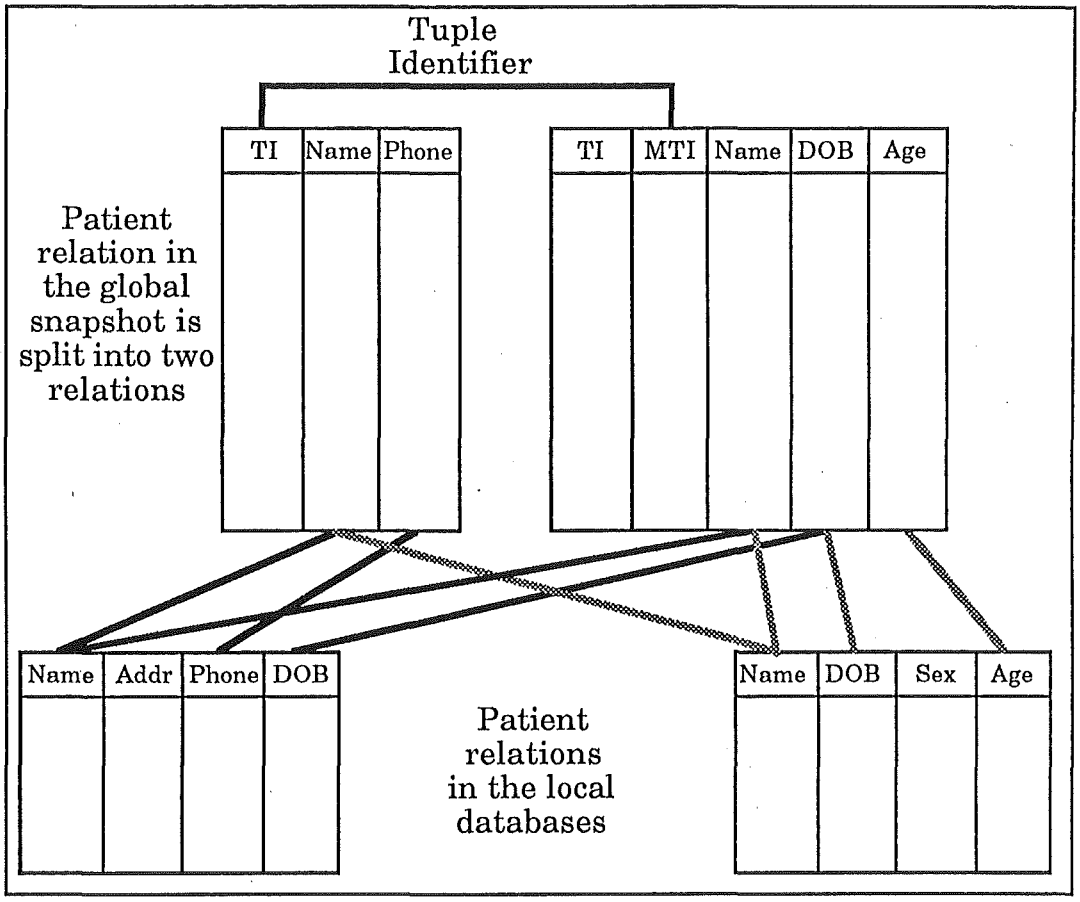

Figure 4.6
Mapping of two local relations at different sites
onto two relations in the global snapshot with non-unique key of the local databases and the global snapshot attribute name

In this example, the **TI** attributes are the unique tuple identifiers that are stored with each tuple in every relation. The **MTI** is a copy of the tuple identifier in the "main" relation of the global snapshot. These will be explained with reference to the following discussion.

The visit relation in the local database.

Name	Phone	DOB
Jones	598-073	12/5/63
Jones	489-207	24/9/68

The visit relation mapped onto the two relations pat-phone and pat-dob in global snapshot.

Relation pat-phone.

Name	Phone
Jones	598-073
Jones	489-207

Relation pat-dob.

Name	DOB
Jones	12/5/63
Jones	24/9/68

In this case there would be a problem of mapping which date of birth value would be mapped to which Jones. This problem is solved by making one of the relations in the global snapshot the main one. All other relations that form part of the map between the local relation and the global snapshot will have an extra attribute. This attribute is the unique tuple identifier of the tuple in the "main" relation. The global snapshot would look like this.

Relation pat-phone.

Tuple Id	Name	Phone
1	Jones	598-073
2	Jones	489-207

Relation pat-sex.

Tuple Id	Main Tuple Id	Name	Sex
21	1	Jones	Male
32	2	Jones	Female

When a join is required, an extra clause is added to the query on the global snapshot. For example a query on the local database.

```
retrieve patient.all
where    patient.name = "Jones"
```

would be mapped onto a query on the global snapshot with the mappings indicated by the lines in Figure 4.6, and the **qualification-loc-to-cent** attribute equal to "pat-phone.tuple-id = pat-sex.main-tuple-id"

```
retrieve pat-phone.name , pat-phone.phone , pat-sex.sex
where    pat-phone.name = "Jones"
and      pat-phone.tuple-id = pat-sex.main-tuple-id
```

This would ensure that the sex attribute values would be mapped onto the correct person. This method would only need to be used on a relation that did allow non-unique key values. It would be very expensive to maintain the tuple identifiers for all the relations mapped onto more than one relation, and would require a lot more extra storage. It is consistent with data models which rely on tuple identifiers such as RM/T [CODD79].

If there were large numbers of queries where these joins were required then thought would have to be given where by the local database schema would be changed to match the global snapshot schema. This would remove the problem of large amounts of extra storage in the global snapshot and the maintenance of the tuple identifiers.

If the key was unique then this each attribute would now need to be added and maintained. This would be the case in most databases so this facility would not be required normally. Having relation with non-unique keys should be avoided as the maintenance of the tuple identifier attributes adds significantly to the processing of this type of mapping.

Possible Extensions to the Mapping System

At present the mapping system can only handle mapping of variables of the same type. A more flexible system would be to allow for different types stored in the local databases to be mapped onto the same type in the global snapshot, and then be able to be mapped back onto different types in the local databases. This would require a function to be developed that would take a value of the type in the local database and convert it into the corresponding type in the global snapshot. This extension would add two extra attributes to the **mapping** relation called **loc-global-function**, which would be the name of the function to convert the value of the local database attribute to the global snapshot type and the other **global-loc-function**, which would do the opposite.

Security and Site Autonomy

Security and site autonomy are important questions in the radiology system. There are large amounts of confidential data to be stored and accessed in the system.

In the Informix database management system [INFO86], there is the facility to restrict access to single relations in the database to specific users. This feature or something equivalent, should be also available in any reasonable database management system. Each user has their own user code, which is used to identify different users. The SNAPS system would have a similar mechanism. As there is a one to many mapping of the local database onto the global snapshot, the same access restrictions can be applied to the relations in the local database. This approach relies on the individual site database administrators restricting access for its users. One site could have no restrictions, and another site could have many levels of restriction. This could case classified information being available to users who have no "right" to have access to it. Therefore the responsibility of security has to be made on a trust basis. Any site that is added to the network must apply restrictions to those relations that are considered sensitive. This provides a high degree of site autonomy, as each site can apply any restrictions it like it does create security problems.

A solution to this problem could be to implement a system similar to that used by the Informix system in the global snapshot controller. This would involve storing a list of user codes from different sites which are allowed access to particular relations.

The global snapshot provides a high degree of site autonomy, as the site of the global snapshot is the only one a local site needs to know about to perform a query. A local database administrator has already said what

data that is stored at this site can be made available to users at other sites, and this is stored in the global snapshot. Therefore there is no possibility of access to other parts of a local database other than those in the global snapshot. There is a loss of site autonomy in that some data stored in the global snapshot can be updated by a user from a site that is not the owner. However, allowing the data to be stored in the global snapshot, is also allowing it to be updatable. Therefore site autonomy has already been lost by that site, not at the time of the update, but when permission of the local database administrator to store the data in the global snapshot was given.

Chapter 5

Query Optimisation

One of the major areas of research in distributed databases today is query optimisation. It is a particularly important aspect as optimisation can reduce the amount of processing and communication needed to perform a query. There are good surveys of the different techniques that can be employed in query optimisation in [CERI84], and [APER83].

In [CELL80], the optimisation techniques that are used in the SIRUS system are presented. The system used in Distributed Ingres is presented in [EPST79]. The system used in SDD-1 is discussed in [GOOD81]. It makes extensive use of semi-join operations. The System R* approach considers transmission costs and local processing costs. [ASTR76]. There are a number of other papers that introduce general methods of optimisation [BERN81a], [CERI83], [MAHM79], [SEGE86] and [YU84].

Most of this research is done to reduce the amount of communication that has to be performed during the processing of a query. This is the underlying approach I have taken with the design of the whole system, not just the query processing component.

[APER83] is entirely devoted to the question of query optimisation and describes various methods of query optimisation which minimise average response time and communication costs. It also discusses the use of data allocation as a means of providing better access to data. This approach is an underlying part of the SNAPS system.

In [APER93], there are three steps that are identified that make up the processing of a query.

- parsing the query
- determining a schedule
- executing a schedule

The question of query optimisation is approached in a similar way in [CERI84]. It splits it up into a number of different areas :

- mapping global queries onto fragments queries
- determining the optimal access strategy
- execution of the optimal access strategy

The question of data allocation is also an important issue and is closely related to query optimisation, as the "determining a schedule" stage uses the knowledge about where the data is stored that is needed to satisfy the query. This subject will be discussed in a separate section later in this chapter. The two are even more closely related if there is dynamic allocation of data.

There are a number of different approaches to query optimisation. The communication costs which is a function of communication time, the average response time and the local processing costs can all be the targets of optimisation. The major target is the communication costs.

Semi-joins have become an important way of reducing the amount of data that has to be transferred between sites, and are used by SDD-1 [GOOD81], as a means of reducing communication. This aspect and the definition of a semi-join will be discussed in a later section.

The use of knowledge of the types of query that are to be performed on the database can be used in query optimisation. In [GARC83], semantic knowledge of queries is used to improve transaction processing. The underlying mapping of the local databases onto a global snapshot in the SNAPS system, is an example of using prior knowledge of the data that will be required in a query, and the way it will be accessed.

Data Allocation

The problem of where to allocate a file or part of it, and its copies, given a known set of updates and retrievals, such that a cost function is minimised, is known as the file allocation problem [APER83]. In most cases the cost function is the total transmission costs. In the distributed database case we can assume that a file equates to a relation.

The major aim of data allocation is to increase the amount of data that is stored at the site that is going to use it the most to reduce the transmission costs involved in a query. A method of determining the optimal allocation of data is presented in [MAHM76].

In the SNAPS system I have implemented a dynamic allocation scheme. The allocation of data is on demand. Initially the global snapshot is defined which is a static allocation of the data all users can have access to. Users perform queries on the local databases, which are mapped onto the global snapshot. Any data retrieved from the global snapshot is stored in the local databases. The data retrieved is stored as a local snapshot of the global snapshot. It is stored in the local databases until it has not been accessed for a user defined number of days when the tuple involved in a snapshot are deleted from the local database. Thus the data is allocated dynamically and on demand.

Semi-Joins : When they are used

A semi-join is a derived operation of relational algebra, which has particular relevance to the optimisation of distributed queries. It has been shown in [ULLM83], that a join operation can be mapped into a number of semi-join programs that produce the same result. The notation used below is the same as that used in [CERI84], which is similar to that used in [ULLM82].

Given a join

"Patient $JN_{pno=pno}$ Visit"

where **pno** and **pno** are attributes of the relations **Patient** and **Visit**, the semi-join program for it would be

Visit $JN_{pno=pno}$ (Patient $SJ_{pno=pno}$ PJ_{pno} Visit)

To understand why semi-join programs are useful, assume that Patient and Visit relations are at different sites. Executing the semi-join involves projecting (PJ) **pno** from the relation **Visit** and sending the result of the projection to the site of the **Patient** relation, where the semi-join (SJ) is performed. Then the result of the semi-join is sent back to the site of **Visit**, and the join (JN) is performed there. The important aspect of a semi-join program is that only a subset of tuples will survive the semi-join operation; only those tuples of **Patient** which will contribute to the result of the final join will be transmitted between the two sites.

In the SNAPS system semi-joins are used to reduce the amount of data that is transferred between the site of the global snapshot, and the site performing the query.

In the next section the type of queries that use semi-joins will be shown, and in Chapter 7 an algorithm that performs a semi-join operation is shown.

Techniques used in the SNAPS System

In the SNAPS system, there is the assumption that there is a database management system that will ensure that queries are performed on the local databases and on the global snapshot. The SNAPS system is an interface on top of a database management system.

In the SNAPS system a number of optimisation techniques have already been applied which reduce the complexity of performing a query.

The global snapshot is the major underlying optimisation in the SNAPS system. It effectively removes all the problems of having to access all the local databases by already performing the union of all the fragments of the relation and storing them at one site. This also gets rid of the problem of having to determine which copy of a fragment to access. In the SNAPS system there is only one copy to access.

The approach I have taken to query processing is similar those proposed in [CERI84] and [APER83].

- check the availability of the global snapshot
- check to see if this query has already been performed on the global snapshot
- parse the query
- map query onto global snapshot
- send mapped query to global snapshot controller
- store resulting data in the local database
- perform the original local query on the local database

There are two types of query that the SNAPS system is designed to handle. The optimising techniques shown in previous sections will be applied to these different cases. An unambiguous clause in a qualification is that one side of the clause is a constant expression. An ambiguous clause is a join clause on two different relations. This will be explained in the following examples.

(i) The simple case is when there is a simple query involving only one relation, and the qualification is not ambiguous. This query returns the patient numbers, their names and ages of all patients called "MacIntosh"

```
retrieve patient.pno,patient.name,patient.age
where    patient.name = "MacIntosh"
```

(ii) The next case is when there is more than one relation is involved, and there is an ambiguous clause in qualification clause. The whole qualification clause is considered unambiguous. This query returns all the visits patient of "Smith" had on the "12/12/87", and there are two similar queries that involve a join operation.

```
[1] retrieve (visit.all)
    where patient.name = "Smith"
      and patient.pno  = visit.pno
      and visit.date   = "12/12/87"

[2] retrieve (visit.all)
    where patient.name = "Smith"
      and patient.pno  = visit.pno

[3] retrieve (visit.all)
    where patient.pno  = visit.pno
```

These queries are essentially the same in that there is a ambiguous clause in the qualification. In the examples [1] and [2], there are unambiguous clauses that can be used to reduce the number of tuples that are sent during a semi-join operation.

To perform the query correctly for this site the data that has been added to the database or modified since the last global snapshot refresh should also be used. This does not need to happen in case (i) as the query involves a selection. The global snapshot controller needs only perform the selection operation on the mapped query. Any new data will be retrieved when the query is passed to the database management system.

In case (ii) it is not so simple. With a join, there could be new or modified tuples that could have an effect on the join operation before the query is processed locally. Therefore there has to be a mechanism that sends the new or modified tuples or a projection of them to the global snapshot, so that they can be used in the join. This mechanism is presented later in this section.

The first operation that is performed when a query is made is to check the availability of the global snapshot. If this is unavailable, then the query can only be handled locally, and the query is passed to the local database management system as if the global snapshot did not exist.

The next step is to check to see if a query on the global snapshot has already been made for this target-list and qualification. The **snapshot-definition** relation is checked to see if the same query qualification, or a query qualification that is more specific than one already performed. This applies to the target-list as well. If the new target-list is a subset of the attributes of a target-list of a previous query, and the query qualification is the same or more specific then the query is assumed to have been performed before, and the results are already stored in the local database.

For example, if a snapshot had a query qualification of the form with the same target-lists for both queries

```
patient.name = "Jones"
```

and new query qualification of the form

```
patient.name = "Jones" and patient.dob = "12/5/34"
```

was checked against the **snapshot-definition** relation, then the second qualification is more specific that the one already performed. Because the results of the first snapshot are already in the local database, there is no need to perform the second query on the global snapshot.

This mechanism is designed to reduce the amount of access on the global snapshot, which reduces overloading of the global snapshot controller. The communication is also reduced as no access to the global snapshot is required.

If an enquiry on the global snapshot is required, then the next operation is to decompose the qualification of the query and the target-list of the query. They are decomposed into sub-queries and sub-target-lists respectively for each local relation referenced in the qualification and target-list. This is similar to the approach taken in [WONG76]. It is also necessary to determine the relations that are mapped onto the global snapshot. If there is a reference to a relation in the qualification or the target-list that is not mapped onto the global database, then any reference in the qualification or target-list to it is discarded until the query is processed by the local database management system. This procedure identifies the local relations that have to be mapped onto the global snapshot. The algorithm for decomposition and an example can be seen in Chapter 7.

From the decomposed form of the query the access strategy can be determined. If there is no join operation in the query, then the query can be processed simply as there is only one table involved.

This is the type of query identified in case (i) above. To execute this type of query, the target-list and the qualification of the query is mapped onto the global snapshot schema. The mapped query is then set to the global snapshot controller, and the query is performed by the controller, and the results are sent back to the SNAPS controller at the local site. The result is then mapped back onto the local database, and then put into the local relation and marked as being part of a snapshot. Then the original query is passed to the local database management system for the query to be performed and the result displayed to the user.

In the second case described above (ii) there is an ambiguous and unambiguous clause in the query qualification, which requires a join operation. The query processing algorithm takes advantage of the fact that global snapshot holds all the data that a local site might want to access

including the data stored at this site. The join operation can take advantage of this fact by performing the join operation on the global snapshot.

However the global snapshot does not have any new or modified data that has been added to this local database during the period between the global snapshot refreshes. Therefore there has to be a mechanism that sends the new or modified tuples or a projection of them to the global snapshot, so that they can be used in the join.

Two semi-join operations are performed, one for each attribute in the join clause and selecting out the data that has been added or modified to the database. This can be determined by using the new and modified flags in the local status attribute and the access rights flag. These semi-joins are projections of the join attributes. If there is an unambiguous qualification clause(s) which references one of the relations in an ambiguous clause, then this clause can be used to restrict the projection resulting in fewer tuples being sent to the global snapshot. Where there is no unambiguous qualification clause for a join attribute, the projection is done with no qualification. For example

```
retrieve (visit.all)
where patient.name = "Smith"
  and patient.pno  = visit.pno
```

would result in one semi-join on the **patient** relation which would project the **pno** attribute with the added selection of patient.name = "Smith". The **visit** relation would just have the **pno** attribute projected with no selection. Where as in the following example there would be no selection on the projection over the **pno** attributes for either attribute.

```
retrieve (visit.all)
where patient.pno  = visit.pno
```

The results of the projections are stored temporarily in the global snapshot along with the user code. The second part of the semi-join takes place when the join is performed by the global snapshot controller, which will reduce the number of tuples that have to be sent back to the local database.

The global snapshot now has all the data that it needs to perform join operation. The mapped target-list and qualification are then passed to the global snapshot controller. After the query has been performed the temporary tuples are also deleted from the global snapshot. Even though these tuples are going to be sent to the global snapshot at the next global refresh, it is easier to delete a tuple which only has one attribute, and add a complete tuple to the global snapshot. The addition in the case of a one to many mapping of the local database to the global snapshot with a non unique key also requires more complex processing.

This approach of sending the projection of the new data to the global snapshot, is a better method than the others possible. This is based on the assumption that the amount of new data added between two global snapshots is always less than the amount stored in the global snapshot. This would not be the case when the system was first started up, and new data was being added to the system, which is a special case.

Once the results of the query on the global snapshot have been added to the local relations the initial query is passed to the local database management system to perform and display the resulting data to the user.

The method present here has dealt with a single join, but can handle any number of joins. All that is required is that a projection of the new and modified data is calculated and sent to an stored temporarily in the global snapshot.

Chapter 6

In Chapter 2 there is a detailed study of the issues relating to snapshots. In this Chapter the extensions and changes I have made to the system of snapshots as discussed in that Chapter are presented. The mechanisms for the creation and refresh of the global snapshot are introduced. The second level of snapshots in the local databases are an important aspect of the SNAPS system. The implementation of read only and updatable snapshots is discussed, and how they are created and refreshed.

Snapshots in the SNAPS system.

The snapshots I have implemented in the SNAPS system, are different from those described by Abida and Lindsay in [ABID80]. The system they defined their snapshots on is significantly different from the system I have proposed. There are a number of different features that I have added that are specific to the radiology system requirements.

All snapshots are based on attributes stored in the local databases. Any combination of attributes from any relations can be retrieved, and include any combination of joins and selections on these attributes. There are no aggregate functions allowed in local snapshots. To achieve the same affect the tuples that are required to compute the function would be requested from the global snapshot, which would then be stored in the local database. Then the aggregate function could be applied tuples in the local database.

The major difference is that their system is based on a normal distributed database, with each site being able to access and update data at all other sites. In the SNAPS system this is not the case. To achieve a high level of site autonomy there is a global snapshot, which stores the select parts of

the database which the other sites are allowed access to. The global snapshot as described in Chapter 4, is a snapshot of selected parts of all the local databases. This is maintained by the global snapshot controller. Any request for data stored at another site is passed the SNAPS controller at its own site which determines if there needs to be any access to the global snapshot. If there is the request is passed to the global snapshot. Thus no local site can access directly data stored at another site.

In Chapter 2 the different ways resources can be distributed, and in this case the database is distributed. There were two cases that were identified as being possible ways of distributing the database for the radiology environment. The first case had the snapshots of the database stored by individual users and maintained by the users. In the second case there was a copy of the database stored at the local site that was access by the users at that site, and is maintained by the owner of the copy.

The two levels of snapshot in the SNAPS system are a cross between these two cases (See Figure 6.1). The users create their own snapshots, but they are maintained by the owner, which in the SNAPS system is the global snapshot controller. The snapshots are also available to all users at the site, not just those who created them.

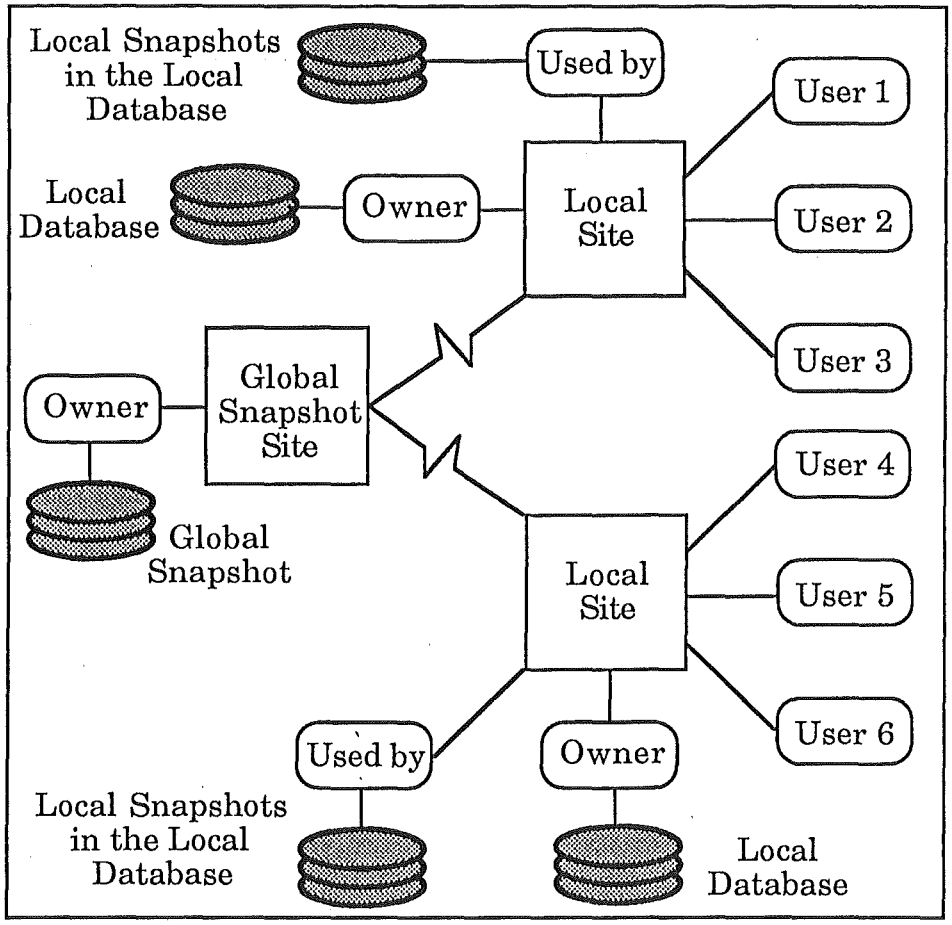

Figure 6.1
Distribution of the Local Databases

This figure 6.1 shows how snapshots have been implemented in the SNAPS system. It shows the distribution the global and local snapshots with the local databases.

The Global Snapshot

The global snapshot schema is defined by the local database administrators. The schema is based on the schemata of the local databases.

The global snapshot is refreshed every day. This was a time frame decided by Dr Kennedy [KENN85], and Dr Feltham [FELT85] as being the most appropriate for their system. This was done for a number of reasons. Firstly the likelihood of a person going to more than one x-ray unit in

Christchurch in one day was very unlikely and the need for other sites to know about information created that day was regarded as minimal. Secondly, for the same reason the need to put new or updated records into the global snapshot quickly was limited. Also the amount of data that is read only in the system means that there will be only very few updates anyway.

Updating daily also allow the communications costs to be reduced as the refresh of the global snapshot can take place outside normal hours and take advantage of the lower rates at off peak times. In the case of the Telecom X.25 charges it is two thirds the normal rate. The aim of the whole system is to reduce the amount of communication to a minimum. If the majority of the communication takes place at off peak rates, the running costs of the system are greatly reduced.

Creating the Global Snapshot

To create the global snapshot the relations in the local databases are mapped onto the schema of the global database. There are three situations that have to be handled where creating the global snapshot. These correspond to the three types of mapping that are possible : one to one, one to many with a unique key, and one to many with a non unique key.

The first case is where there is a one to one mapping. In this case the mapping is a projection of the local relation onto the global snapshot relation. To create the global snapshot, a projection of the local relation is appended to the relation in the global snapshot relation.

The second case is similar to the first. In this case a projection of the local relation is performed for each relation that is mapped in the global snapshot, and the results appended to the global snapshot relation.

Then a retrieval is performed on the mapped relations and all tuples that are retrieved are put into the global snapshot. There is no qualification to the global snapshot definition.

This approach works for a one to one and a one to many mapping of a local relation onto the global snapshot, if the local relation has a unique key. In the one to many case there is a retrieval for each of the relations mapped onto. There is no extra processing required to put the tuples from the local relation into the global snapshot. The algorithm is defined in Chapter 7.

The case of a local relation with one to many and a non-unique key requires a different approach to place the tuples in the correct relations, to be able to match them up when a join has to be performed. In the mapping system a main relation is defined from the global snapshot relations involved in the map. The the projection onto this main relation is performed. Tuples then have to be added individually to the main relation. After each tuple is added, the tuple identifier has be retrieved from this relation. Then the projection of the other relations mapped onto the global snapshot is performed for this tuple, and the main relation tuple identifier is added to this projection, and then stored in the global snapshot relation.

The algorithm to perform the creation of the global snapshot is defined in Chapter 7.

Refreshing the Global Snapshot

The refresh of the global snapshot is performed each day. To perform this the local databases must all be available. To perform the refresh correctly no access to the local database is allowed during the period of refresh. As stated before the timing of the refresh would be such that it would take place when the least inconvenience would be caused.

The global refresh operation consists of the following tasks :

- delete tuples from local snapshots that have been modified by other sites
- commit updates of local snapshots to the global snapshot
- append new data to the global snapshot relations
- delete tuples from the global snapshot
- reset all the update flags back to there preset values

The first step is to delete all tuples from the local relations where the access rights flag is set to locked. All the sent-to flags are set to false except the site had had acquired the lock. This operation is part of the local snapshot refresh mechanism, but has to be performed before the access rights flags are reset.

The next step is commit updates to the local snapshots to the local databases. In the case of the unique key relations the non-key fields can modified from the values stored at the local site where the lock was acquired from.

In the first step each site sends all the new data that has been created at that site. The database management system used by the global snapshot is assumed to capable of notifying users if multiple values of the same key are attempted to be added to the global snapshot. This problem was addressed in Chapter 4, and a possible solution was proposed which required that a site code be added to each key value to achieve uniqueness. This problem only occurs if the global snapshot has been unavailable when new tuples were added to the local database. During this time, it would not be possible to check to see if tuples already existed for this key value.

If a tuple is marked as deleted in the local database does not mean it will be deleted from the global snapshot. If the sent-to-flag in the global snapshot indicates that this tuple has been sent to any site, then this tuple

can only be deleted from the local database. Because of the high level of site autonomy, no site can delete tuples stored at another site even if the tuple is owned by that site. The sent-to-flags for each site shows which sites have a copy of this tuple, only when all the flags are false can the it be deleted from the global snapshot.

The last operation is the resetting of the access rights on each tuple to the preset value for this relation. This can be read only for read only relations or read/write for updatable relations.

The Local Snapshots

When a SNAPS controller decides that it needs to access data stored at other sites, it maps it query onto the global snapshot and sends it to the global snapshot controller. This mechanism is discussed in Chapter 5, and the algorithm to perform a query on the global snapshot is presented in Chapter 7.

The data that is returned from the global snapshot controller is called a local snapshot. If the target-list and qualification of the snapshot involves more than one local relation then the result of the snapshot will be placed into those relations that are mapped from the global snapshot onto the local database.

One of the major difference with the definition of snapshots in the SNAPS system, and that in [ABID80], is that they are stored as part of the relation they are part of. Tuples that are part of a snapshot can be distinguished from the rest of the relation by the fact that the tuple type in the status attribute is set to snapshot as opposed to local data. This is important, as in [ABID80], it was stated that the source relations of a snapshot should be able to continue their own independent evolution. This rule is not

contravened by storing the snapshots in the local relations, as the snapshots are independent of the tuples in the source relation.

For each relation referenced in the target-list or the qualification of a query and mapped onto the global database, any tuple that is required to evaluate the snapshot at the local sites will be retrieved and stored in the local database, not just those attributes mentioned in the target-list of the qualification that are required.

This is done for a number of reasons. In the radiology system, the database schema is split up into relations that reflect the operational requirements of the system. The relations are based on how the data is to be accessed by the application. When a query is performed in the application, just about always all the attributes of a relation are to be retrieved. (See Appendix G for a detailed description of the radiology system schema).

If this is not the case then the local database schema could be changed to reflect the access requirements of the user, by splitting the local relations up into smaller relations, that map more directly onto the global snapshot schema. This would reduce communication costs, by not retrieving unwanted data, and also reducing storage requirements, and null attributes would not be stored in tuples that are part of a snapshot.

The result of a snapshot is stored in the relation so that if a user accesses this relation then they will also retrieve the tuples that are parts of a snapshot. After a tuple has been added to the local database relation, it essentially become part of the local database, and can be accessed the same way as any local data.

In a sense, the snapshot mechanism I have designed is a cross between multiple copies and traditional snapshots. They are snapshots in the fact

that they are a state of the database at a particular time, and where the relation is of read only nature to all sites acts exactly like snapshots as described in ABID80. They are also are like snapshots in that they are automatically refreshed by the global snapshot controller. They are like multiple copies as they can accessed as individual tuples as part of a query on the local database, and not as part of the snapshot explicitly, and that a snapshot of the global snapshot can be stored at more than one local site.

Another feature of my snapshots is that if they are not accessed, for a specified number of days, defined by the local database administrator, then they will be deleted from the local database. This was considered important as there were limits on the amount of data that could be held at the local sites on their file servers.

Implementing Local Snapshots

There is a **snapshot-description** relation that stores the actual target-list and query qualification that are used to produce the snapshot. The detailed format of this is shown in Appendix D. Each time a snapshot is created, a new snapshot number is allocated, and the definition stored in this relation.

There is also a **snapshot-definition** relation, which stores the snapshot number, the local relation name, and the list of tuples that make up a snapshot.

Every relation in the local database that is mapped onto the global snapshot has an extra attribute added onto it (See appendix A for a full description). As snapshots in this system are added to the local relation, and not stored separately, there has to be some means of identifying tuples that are parts of snapshots in the local relation.

This extra attribute contains the tuple type, which indicates either local data or a snapshot.

It also indicates which access rights this site has over this tuple. If the relation is a read only relation then the access rights will be read only for all tuples that are part of a snapshot, while if the tuple is a local one, then the access will be read/ write. If the relation is read/write then the access rights can be either unknown, read only or read/write.

It will be known if this site has not tried to acquire the lock for this tuple from the global snapshot controller. It will be read only if this site has tried to acquire the lock for this tuple from the global snapshot controller, and the relation type is read/write, but another site has already acquired it. It will be read/write if this site has successfully acquired the lock from the global snapshot controller. If the access rights indicates that the tuple has been deleted then, the tuple will be deleted from the local database when the next global snapshot refresh is done.

It also indicates if the data is new. This means that the tuple has been added to the local database since the last global snapshot refresh. This is used when the global snapshot refresh is done.

Another flag which is similar is the modified flag. This indicates if the tuple has been modified since the last global snapshot update. If the relation is a read only one then this flag is used to indicate that this site has updated its own data. If the relation is updatable then the access rights flag will indicate if the tuple has been updated, as the lock will have to have been acquired successfully. This flag will then not be used for read/write relations, only for read only ones to indicate local updates.

Creating Local Snapshots

A local snapshot is created when an enquiry on the global snapshot is made. When the results of a enquiry are loaded into the local relations the tuple identifiers of all the tuples must be added to the **snapshot-definition** relation. When a tuple as added to the local relation, the just-copied variable is set to a unique value defined by the SNAPS controller. After all tuples have been added a query is performed on the local relation(s) to retrieve the tuple identifiers for those tuples that have been just copied. The just copied variable is then set to false for those tuples.

The sent-to-flag for this site in the global snapshot status attribute will also be set for the tuples that have been sent to this site.

Refreshing Local Snapshots

The differential approach is used to refresh the local snapshots. The information stored in the status attribute of the global snapshot relations has been designed to be used by the refresh algorithm. All tuples that have been added to the database or been modified will be stored in the global snapshot with all the sent-to flags set to false. All the local snapshots at all the sites have to be re-executed to see if any of the new data or modified data has been added to the global snapshot.

The first steps of the global snapshot refresh is to delete from tuples in local snapshots that have been updated. This is part of the local snapshot refresh mechanism. This done so that tuples in the local snapshot that satisfied the query before they are updated are not left in the snapshot if the modification would means that they do not know fit the snapshot qualification.

The next step is to re-execute the snapshot descriptions. No query optimisation is needed at the local site as all the data is in the global snapshot. The snapshot description is mapped onto the global snapshot as per a normal query and passed to the global snapshot controller. The only tuples that have to be sent back to the local site will be those tuples that are retrieved and have their sent-to flag set to false. These flags will be set to true for those tuples that are sent. this will retrieve all the new tuples and those tuples that have been modified and still meet the snapshot qualification. The re-execution might not involve the sending of any tuples at all. This is the advantage of using the differential refresh approach as opposed to the deleted and re-calculate approach, which would require all tuples selected in the local snapshot to be sent. Both approaches require large amounts of local processing by the global snapshot controller, so the reduction in the communication costs by using the differential approach is significant.

Updatable Snapshots

One major departure from the snapshot definition is that ability to update tuples that are part of a local snapshot. One of the reasons that snapshots were proposed for distributed databases, is that they reduce communications, the costs of maintaining multiple copies and the costs of making transactions consistent. However the radiology system differs significantly from the generalized distributed database environment. The architecture of the distributed database is quite specific to this system, and the requirements for access and update ability can not be considered as the general case. Most of the relations in the local database can not be updated by other sites. In this case, the snapshots are similar to traditional snapshots definitions.

There are some relations in this system where limited update ability to other sites would make the system more usable. For example the patient relation, is probably the least static of all the relations in the context of the radiology system, but in a more general system would be regarded as quite static. Combined with the global snapshot site controlling access and updates on all the data stored at other sites, the costs of providing update ability is reduced significantly.

The requirements for updating in a general distributed database environment are expensive in communication costs, and co-ordinating the transaction at all the sites. With multiple copies this problem is compounded even more.

The method I have proposed does not impose large amounts of communication costs, and only involves the central site and the site wanting to perform the update so that the are not the same level of complexity of co-ordinating the update transaction. The concurrency control method is defined in detail in the next section.

Concurrency Control for the Updatable Snapshots

There have been a number of systems proposed to handle concurrency control in distributed databases. A good summary of concurrency control is found in [CERI84], [PAPA84], and [ULLM82]. A number of surveys have also been done on comparing different strategies [BADA80], [BERN83], [BERN84], [DAVI84], [MILE80], and [TAY85].

There are the methods that are similar to locking in centralised database systems. These use the two phase locking protocol, which is described in [ULLM82]. In [GARC79] a concurrency control method is proposed which uses centralised locking controller, which is relevant to the SNAPS system.

In [ESWA76] use the use of predicate locks is discussed. Predicate locks lock selected parts of a database by defining a query qualification to parts of the database which match parts of the database. This system is considered for the SNAPS system, with the possibility of using the snapshot qualification as the predicate. However the size of the lockable objects are dynamic and could result in large parts of the global snapshot being locked that were not even updated, which could result in an increase in the number of aborted transactions. When a lock is requested in the SNAPS system it is done in the form of a query, but the query is very specific and will only ever return one lock at a time.

Other methods proposed by Bernstein and el. for use in SDD-1, use timestamps as the method of ensuring serialisation in distributed transactions [BERN80a] and [BERN80b].

Another method is the majority consensus system, where by the controlling site must have a majority of "update ok" votes from the sites at which the multiple copies are stored, to perform the update.[THOM79].

The system used in Distributed Ingres is presented in [STON79], which used the primary copy method of locking [ULLM82], to achieve concurrency.

The method I propose to use in the SNAPS system is a combination of primary locking method [ULLM82], and optimistic concurrency control. [KUNG81] and [SCHL81].

Optimistic concurrency is based on the assumption, that conflicts among transactions will be rare, and that concurrency need not be based on locking, but can be based on backup as the main mechanism.

In the radiology environment, the chances of the same person going into different places to have x-rays, and the users at those sites wanting update

the same information is considered to be very rare to the point where it is zero. This is why optimistic concurrency control is considered appropriate for the SNAPS system.

Optimistic concurrency control was designed to be used in an environment where updates were committed when the transaction was performed. In the SNAPS system, updates will not be committed to the global snapshot, and the changes propagated to other sites which have this data stored as part of a local snapshot, until the global snapshot refresh.

This is where the locking aspect of the concurrency control scheme for the SNAPS system comes in. The primary copy method assumes that one copy of data is the primary copy. All request to update the data must go to the primary copy, which holds the lock for this data [ULLM82].

When a user requests data from the global snapshot there is no assumption made about whether the data will be updated. Thus no lock is placed on the data when it is read. Data can be read from the global snapshot which has been locked as the basic assumption about snapshots is that they store the state of the database as of a previous time and that users will tolerate out of date data.

Only when a user wants to update some data are the locks acquired from the global snapshot. This uses the primary copy approach as the global snapshot controller holds all the locks for all the copies stored in local snapshots. Only if all the locks required by a user are acquired, can the SNAPS controller update the local snapshot.

The global snapshot is not updated at the time the locks are acquired. The updates are only committed to the global database when the global snapshot refresh is performed.

The SNAPS system allows for updatable snapshots which conflicts with the semantics of snapshots as proposed in [ABID80]. This is because the SNAPS system offers the features of updatable multiple copies and snapshots combined. In [ABID80] modifications to the source relations are reflected in the snapshot when it refreshed. The local snapshots can be considered to be the source relations of the global snapshot, and changes to the local snapshots are reflected in the global snapshot when it is refreshed. The global snapshot can now be considered to be the source relations of the local snapshots. Thus changes in the global snapshot are reflected in the local snapshots when they are refreshed.

Thus the semantics of snapshots have not been broken if you look at the overall system. Where my system is different is this is that the actual local snapshot is updated and in the process of refreshing the changes are made to the global snapshot. However if the cyclic relationship between the local snapshots and the source relations and the global snapshot is considered, then when a local snapshot is updated it can be considered to be part of the source relation of the global snapshot.

Thus updatable snapshots in the SNAPS system do not contravene the snapshot semantics proposed by Abida and Lindsay.

Chapter 7

The Algorithms

This chapter defines the algorithms that are used to implement the SNAPS system. The two components of the SNAPS system are : the local SNAPS controller and the global snapshot controller. The following functions of both can been identified from the previous chapters.

The functions performed by the local controllers are :

- perform local queries and enquiries on the global snapshot
- request locks for tuples from the global snapshot controller
- check for non-mapped attribute update of tuples in a local snapshot
- refresh local snapshots
- delete local snapshots from the local database that have not been accessed for a user defined number of days

The majority of operations that the local controller is required to perform are related to processing queries. It also has to request locks for tuples that are being modified. Checking for the problem identified in chapter 4 regarding non-mapped attribute updates of tuples in a local snapshot also has to be handled.

The functions of the global snapshot controller are :

- create global snapshot
- refresh global snapshot
- manage locks on updatable relations
- perform queries and return data to local sites

The primary function of the global snapshot controller is to perform queries that compute local snapshots. It is also required to manage locks on the updatable relations. The controller is also responsible for

controlling the global snapshot refresh. This is performed in conjunction with the local snapshot refresh.

Algorithms to perform all the functions described above are presented. These algorithms were implemented in C [KERN86], and using the EQUEL [KALA86] pre-processor to QUEL. They are presented here in a Pascal type language, and a combination of Ingres statements. Examples of the use and environment in which they are used have been presented in previous chapters.

Algorithm to Perform a Local Query

The algorithm to perform a query at a local site is split up into a number of parts. The following is an overview of the algorithm and the parts that have to be defined to a lower level are present after this. The first thing that is checked is to see if query on the global snapshot has already been performed for this query. If a query has then no access to the global snapshot is required, and the query can be performed on the local database.

If a query on the global snapshot has not been performed for this query, then, the next step is to check to see if the global snapshot is available. If it is not available the query can only be performed on the local database.

In the normal situation the global snapshot is available. The next step is to decompose the query to determine the type of query. A definition of the decomposition algorithm can be found later in this chapter. The two types of query are ambiguous, which have join clauses in their qualification, and unambiguous queries which involve only a selection on one relation. The requirements of the different types of query are defined in later sections.

After the query had been decomposed the mapped query is sent to the global snapshot controller, and the data received from the global snapshot has been added to the local database, the query can be executed on the local database which will display the data that the user wanted to see.

```
if not performed (target-list, qualification) then begin
   if global-snapshot-available then begin
      decompose (target-list, qualification)
      if query = ambiguous then
           ambiguous   (target-list, qualification)
      else unambiguous (target-list, qualification)
         end
      end
   end
perform-local-query(target-list, qualification)
```

The **performed** function takes a target list and a qualification and returns true if on checking theses against descriptions of queries that have already been performed on the global snapshot, which are stored in the **snapshot-description** relation, an exact match can be found or if this query is more specific in either the qualification or the target list than one already performed.

The **global-snapshot-available** function checks to see if the global snapshot is available to be used. It might be in the middle of refreshing the snapshots, in which case the query would be aborted as the local databases are unavailable at this time as well. The communication network could be down. This function would attempt to create a connection with the global snapshot and if this fails the function would return false otherwise true.

The **decomposition** procedure is defined in a following section. The variable **query** is set in the decomposition procedure to the type of query to be performed, which can be ambiguous or unambiguous.

The last procedure called is **perform-local-query**. This procedure passes the query directly to the local database management system to perform. When the query has been performed the **number-of-days-since-last-access**

variable needed to be updated for those tuples that were accessed and that are parts of a local snapshot. When the query is being performed, there should be a mechanism that detects which tuples have been accessed. Then after the query has finished the list of tuple accessed can be matched against the **snapshot-description** relation and the **number-of-days-since-last-access** variable can be reset.

A method of solving this problem is to perform the query, with the same qualification, but retrieve only the tuple identifiers of the tuples match by the qualification. This would result is extra local processing, but if the number of tuples that were retrieved and the queries generally simple the amount of extra processing required would not be significant. This **number-of-days-since-last-access** variable should not be updated if it has the value 0 as this indicates that the tuple has been locked into the local database as a non-mapped attribute in this tuple has been updated.

The Unambiguous Query Algorithm

The unambiguous procedure takes the target list and the qualification of the query. It then maps the query onto the global snapshot, which is passed to the global snapshot controller. No pre-processing needs take place for this type of query. Then this process is put into the wait state while the query is computed at the global snapshot.

There is a limit on how long a process should wait before it gives up, and assumes that the global snapshot is now unavailable. In the normal course of events the global snapshot controller will send the result of the query back in a reasonable time. This data is then put into the local database.

```
procedure unambiguous (target-list, qualification)

begin
mapped-query = map (target-list, qualification)
send-query (mapped-query)
wait (time-limit)
if timed-out then
   abort
else begin
   snapshot = new-snapshot(target-list, qualification)
   append-to-database (snapshot,target-list,result)
   end
end
```

The **map** function takes a target list and qualification and maps the attributes onto the global snapshot. It returns the mapped target-list and qualification as a single string.

The **send-query** procedure takes a query and sends it to the global snapshot controller to perform. The **wait** procedure waits for a set amount of time. If this process is not interrupted before the time limit, the **time-out** variable is set to true, otherwise it is false. The **new-snapshot** procedure takes the target list and qualification of the query and defines a new snapshot. A new unique **snapshot-number** is returned.

The results are returned and are put into the local database by the procedure **append-to-database**. This procedure takes the snapshot number, the target list and the data sent by the global snapshot controller. It then maps the result onto the target list and appends the tuples to the local relations. It sets the status attribute variables, the tuple type to snapshot and the access rights to the default value for that relation. The just copied flag is also set to a value defined by the SNAPS controller. This is so the tuple identifiers can be retrieved and added to the **snapshot-description** relation to define this new local snapshot. The **snapshot** variable is a unique snapshot number, that is used when tuples are added to this relation.

The Ambiguous Query Algorithm

The ambiguous procedure like the unambiguous one takes a target list and qualification. The only difference between this and the unambiguous algorithm is that there is a call to the **send-new-tuples** procedure. This procedure is discussed in the Section on Semi-Joins as part of a Local Query.

```
procedure ambiguous (target-list, qualification)

begin
mapped-query = map(target-list, qualification)
send-new-tuples (attr)
send-query(mapped-query)
wait(time-limit)
if timed-out then
   abort
else begin
   snapshot = new-snapshot (target-list,qualification)
   append-to-database (snapshot,target-list,result)
  end
end
```

More data will be returned with a join operation, as the whole tuples of the join relations are returned, not just the join attributes. The whole tuples are returned so that the local snapshot can be used, and also the individual tuples can be accessed as on their own. It would not be very useful just to store the join attributes, so it was decided to store the whole tuple. This is relevant to the radiology system in that when a relation is accessed the whole tuple is requested. In normal snapshots the join would not have to be performed as the result of the query is stored in a separate relation. The joins will have to be performed again during the local query processing, but the local processing costs were considered so be less important that the the communication costs. The approach I have take also increase the availability of the data to the user.

The Query Decomposition Algorithm

This algorithm decomposes the query into sub queries, and determines a list of relations referenced in the qualification and target list that are mapped onto the global snapshot. It also determines the type of query, which is either ambiguous, or unambiguous.

This algorithm splits up the qualification into sub queries. It identifies each clause as either being ambiguous (e.g. patient.pno = visit.pno), or unambiguous (e.g. patient.sex = "male"). For each clause of the qualification that is unambiguous, the table involved in the clause if found. This clause is then added to the list of unambiguous clauses, stored in variable **clauses**, already found for this table. If the clause is ambiguous then it added to a list of other ambiguous clauses.

```
while not end-of-qualification do begin
   clause = get-next-clause(qualification)
   if is-ambiguous(clause) then begin
      ambiguous-clauses = ambiguous-clauses + clause
      end
   else begin
      table = get-table(clause)
      clauses(table) = clauses(table) + clause
      end
   end
if null(ambiguous-clauses) then
   query = unambiguous
else query = ambiguous
   end
```

The function **get-next-clause** returns the next clause from the qualification supplied as the argument. The function **is-ambiguous** returns true if the clause is of the ambiguous type as defined above. The function **get-table** takes a clause and returns the relation that is referenced in the clause. The function **null** takes a list and returns true if the list is empty.

The next part of the algorithm splits the target list up to determine which local relations have been referenced. This is done to determine which

relations have to be mapped onto the global snapshot, and for those relations that do not mapped any reference in the target list can be discarded until the query is performed later by the local database management system.

```
while not end-of-target-list do
     item = get-next-item(target-list)
     table-name = get-table(item)
     if is-mapped(table-name) then
          mapped-tables = mapped-tables + table-name
   end
```

The function **get-next-item** gets the next reference to a relation.attribute combination. The function **is-mapped** returns a boolean if there is a reference in the **mapping** relation to the table-name supplied as an argument. The variable **mapped-tables** is a list of the local relations that are reference in the target list and that are mapped onto the global snapshot.

After the qualification has been decomposed, if the list of ambiguous clauses is null, then the query does not involve any join operations. If the query does involve ambiguous clauses or join clauses then the ambiguous clauses have to be decomposed more to determine the relations involved in the join. The relations are added to a list of all the relations involved in the query that are mapped onto the global snapshot.

```
for each clause in ambiguous-clauses begin
   table = first-table(clause)
   if is-mapped(table) then begin
      join-attr = first-attr(clause)
      if not in-list(join-attr, attrs(table)) then
         attrs(table) = attrs(table) + table.join-attr
      if not in-list(table, mapped-tables) then
         mapped-tables = mapped-tables + table
      end
   table = second-table(clause)
   if is-mapped(table) then begin
      join-attr = second-attr(clause)
      if not in-list(join-attr, attrs(table)) then
         attrs(table) = attrs(table) + table.join-attr
      if not in-list(table, mapped-tables) then
         mapped-tables = mapped-tables + table
      end
   end
```

The function **first-table** returns the relation referenced on the left hand side of the join clause, and the function **second-table** the relation on the right hand side. The function **first-attr** returns the attribute referenced on the left hand side of the join clause, and the function **second-attr** the attribute on the right hand side. The function **in-list** returns true if the first argument is in the list supplied as the second argument.

The query has now been decomposed to a point where control is passed back to the local query processing algorithm.

Semi-Joins as part of a Local Query

This algorithm takes a list of the relations in **mapped-tables** that are involved in the query and are mapped onto the global snapshot. It checks see if the **attrs** list for each table is null. This list was computed during the query decomposition stage. If the list is not null then the attributes in this list are involved in join operations. To reduce the number of tuples that are to be sent, the unambiguous or selection clauses are also applied to the query. Then the results of the projection on the relation **table** with the unambiguous qualifications in **clauses** and the added clause of "**table**.status.new-data=true". This clause is added as only those tuples that have been added since the last global snapshot are required to be sent to the global snapshot. The other are already in the global snapshot.

```
procedure send-new-tuples
;
; This is where the semi-join operations are performed
; that put the values of join attributes that are new,
; into the global snapshot
;
for each table in mapped-tables begin
   if attrs(table) <> null then begin
      ;
      ; perform local query on the attributes
      ; in attrs(table) and send results to
      ; the global snapshot controller
      ;
      send-result (
         attrs(table),
         "clauses(table) and table.status.new = true",
         temporary)
      end
  end
```

The **send-result** procedure takes a target list and a query qualification. It also takes a variable which is either temporary or permanent. Temporary indicates that the tuples being sent are for use in a join, and can be deleted from the global snapshot after the join has been computed. Permanent indicates that the data is to be stored in the global snapshot until it is deleted by a user. The results of this query grouped together to reduce the amount of communication and then sent to the global snapshot controller.

This algorithm show the use of semi joins in the SNAPS system. The other projection and selection of the semi-join are performed by the query processor in the global snapshot controller and the results sent back to this site.

Requesting Locks from the Global Snapshot Controller

In chapter 6 the concept of updatable snapshots was introduced. The mechanism that is used to ensure consistency is a combination of the primary copy locking protocol, and the optimistic concurrency control mechanism. The algorithm to get lock is very simple. When a user wants to update a tuple in the local database, there have to be a number of check that have to be made to see if the modification can be made.

Firstly, the type of the tuple is checked in conjunction with the type of the relation. The only time a lock needs to be acquired is if the relation type is read/write and the access rights are unknown. In all other cases with read only relations users can only update their own data, and no locks are needed to be requested. For read/write relations if the state is either read only or read/write then the access rights have already been requested. The first case with the rights set to read only, indicates that the access rights were acquired by another site before this one attempted to request them, and therefore this site can not update this tuple until after the global snapshot refresh when the lock is relinquished by the other site. In the second case the rights have been requested and granted to this site. This site has the update rights for this tuple until the global snapshot refresh.

This algorithm implements the above strategy for lock management. The request come in form of a query qualification which defines the tuples that are wanted to be updated. In mosts cases the key of the relation should be used as the basis of the qualification, as only one lock can be requested at a time.

```
function update-request ( table, qualification ) boolean

begin
begin transaction
   retrieve (table.status.type, table.status.rights)
      where qualification
   if table.status.access.rights = unknown then begin
      ; the rights have to be requested
      ; from the global snapshot controller
      mapped = map(table.status.rights, qualification)
      access-rights = send-request(mapped,qualification)
      update (table.status.rights = access-rights)
       where qualification
      if access-rights = read/write then
         update-request = true
      else update-request = false
      end
   else if table.status.access.rights = read/write then
         update-request = true
      else if table.status.access.rights = readonly then
            if table.status.type = local then
               update-request = true
            else update-request = false
               end
            end
         end
      end
end transaction
```

The begin and end transaction statements are so that the requests for locks and updating of then at this site can be seen as one transaction, so that two users at the same site can not attempt acquire the same locks. These construct are used in Ingres and it is assumed that there is a similar mechanism to group transactions into being considered as one single transaction, in the database management system to manage the global snapshot.

Algorithm to Check for Non-Mapped Attribute Updates

This algorithm is run at the time of the local snapshot refresh when the **days-since-last-access** variable is updated. The algorithm checks to see if any tuples that are parts of snapshots have attributes that are not mapped onto the global snapshot have values other than null for strings and 0 for numbers. All the local relation names found in the **mapping** relation are

checked. This algorithm assumes that access is possible to the system relations which store the names of relations, attributes and their types.

The tuple identifiers of tuples which have had non-mapped attributes modified are retrieved, and then used to match the tuple identifiers stored in the **snapshot-definition** relation. The **number-of-days-since-last-access** variable is set to 0 which is the locked state for each tuple that has been modified.

```
for each relation in mapping-relation
   check-qualification = get-all-attributes(relation)
   retrieve (relation.tuple-id)
   where check-qualification
     and status.tuple-type = snapshot
   for each tuple-id in result
      update snapshot-definition.number-days-since = 0
      where tuple-id = snap-tuple-id
      end
   end
```

The **get-all-attributes** procedure accesses the system relations and returns a qualification which is of the form "relation.age <> 0" for an attribute that is a number and "relation.notes <> " ", for each relation that is not mapped onto the global snapshot.

Algorithm to Refresh Local Snapshots

This operation is not very complex but requires a lot of local processing. This operation should take place at the same as the global snapshot refresh, but started after this refresh

All the snapshots have to be re-executed at each site. As the global snapshot is completely up to date no data needs to be sent to the global snapshot as is required when a query is performed. Therefore the query can be passed straight the global snapshot controller. Only those tuples that have their sent-to flag set to false need to be sent to the local database. This clause "status.sent-to(this-site) = false" is added to the qualification that is sent to the global snapshot controller so that tuples that have been

already sent to this site are not sent again unnecessarily. The tuples that have been modified or added to the global snapshot have had their sent-to flags set to false so that this refresh algorithm will pick up any changes to the global snapshot

```
procedure local-snapshot-refresh

begin
for each snapshot-definition in snapshot-definition
   begin
   mapped-query = map (target-list, qualification
                +"status.sent-to(this-site) = false")
   send-query (mapped-query)
   wait (time-limit)
   if timed-out then
      abort
   else begin
      append-to-database (snapshot,target-list,result)
   end
end
```

This algorithm uses the same procedures as the local query algorithm. the **snapshot** variable is set when the snapshot definition relation is accessed.

Algorithm to Deleted Local Snapshots

This algorithm would be run before the local snapshot refresh, so that the snapshots that are due to be deleted on this day are not refreshed unnecessarily. It is also run before the global snapshot refresh, so that tuples that have been deleted from all sites can be deleted from the global snapshot.

This algorithm uses the information stored in the **snapshot-description (SD)** relation. It performs a query on the **number-of-days-since-last-access** attribute comparing this with the user defined maximum number of days (**max-days**) This can be up to a maximum of 256 and would depend on the amount of space the local fileserver has, and other needs of the users. This algorithm uses the **mapping** relation to determine the list of relations that could have snapshots in them

```
for each relation in mapping-relation
   delete from relation
    where relation.tuple-id = SD.snaps-tuple-id
      and SD.number-of-days-since-last-access > max-days
```

Algorithm to Load the Global Snapshot

It is assumed that the relations that make up the global snapshot have been created before the loading takes place. This algorithm uses all the **mapping** relations stored at the local sites to determine which attributes are to be mapped into the global snapshot.

There are three situations that have to be handled where loading the global snapshot. These correspond to the three types of mapping that are possible : one to one, one to many with a unique key, and one to many with a non unique key.

The first case is where there is a one to one mapping. In this case the mapping is a projection of the local relation onto the global snapshot relation. To load the global snapshot, a projection of the local relation is appended to the relation in the global snapshot relation.

The second case is similar to the first. In this case a projection of the local relation is performed for each relation that is mapped in the global snapshot, and the results appended to the global snapshot relation. The mapping is split up into the individual relations in the global snapshot, so there is a projection for each relation.

Then a retrieval is performed on the individual projections of the local relation and all tuples that are retrieved are put into the global snapshot relation.

The case of a local relation with one to many and a non-unique key requires a different approach to place the tuples in the correct relations, to be able to match them up when a join has to be performed. The algorithm

to perform the this type of loading was never developed fully as this type of mapping was no used in the radiology system I was modelling. The whole relation with out being mapped is sent to the global snapshot controller so that all the processing can be done there. In the mapping system a main relation is defined from the global snapshot relations involved in the map. The the projection onto this main relation is performed. Tuples then have to be added individually to the main relation. After each tuple is added, the tuple identifier has be retrieved from this relation. Then the projection of the other relations mapped onto the global snapshot is performed for this tuple, and the main relation tuple identifier is added to this projection, and then stored in the appropriate global snapshot relation.

```
procedure load-global-snapshot

begin
for each relation in mapping-relation
   mapping = get-map(relation)
   split-map(mapping)
   if unique-key(relation) then begin
      for i = 1 to number-relations
         send-result(split(i),null,permanent)
         end
   else begin
      send-result(mapping,null,temporary)
      (see previous text for description
       of approach that would be taken)
      end
```

The **get-map** function gets a list of all the attributes, and relations that are mapped onto the local relation passed as the parameter.

The **split-map** procedure splits the value of the **mapping** variable up into the projections of the local database relation for each of the relations in the map. The key to the local relation is added to the non main relations in the map as the key is not mapped for each of the global snapshot relations in the **mapping** relation. The variable **number-relations** is set to the number of relations that are mapped by this local relation.

The **unique-key** function returns true if the relation supplied as an argument has a unique key to identify each tuple. The **send-result** procedure was defined in the Section on Semi-Joins as Part of a Local Query.

Algorithm to Refresh the Global Snapshot

The refresh of the global snapshot is performed each day. To perform this the local databases must all be available. To perform the refresh correctly no access to the local database is allowed during the period of refresh.

The global refresh operation consists of the following tasks :

- delete tuples from local snapshots that have been modified by other sites
- commit updates of local snapshots to the global snapshot
- append new data to the global snapshot relations
- delete tuples from the global snapshot
- reset all the update flags back to there preset values

The first step is to delete all tuples from the local relations where the access rights flag is set to locked. All the sent-to flags are set to false except the site had had acquired the lock. This operation is part of the local snapshot refresh mechanism, but has to be performed before the access rights flags are reset.

The next step is commit updates to the local snapshots to the local databases. In the case of the unique key relations the non-key fields can modified from the values stored at the local site where the lock was acquired from. In the case of the non unique key relations the whole tuple should be deleted and then re-inserted into the global snapshot relations.

In the first step each site sends all the new data that has been created at that site.

If a tuple is marked as deleted in the local database does not mean it will be deleted from the global snapshot. If the sent-to-flag in the global snapshot indicates that this tuple has been sent to any site, then this tuple can only be deleted from the local database.

The last operation is the resetting of the access rights on each tuple to the preset value for this relation.

This algorithm is designed to be used with unique keys. The case of non unique keys requires the values all of the attributes in the map to be sent as part of the qualification, rather than just the key in the case with unique keys.

```
procedure global-refresh

begin
;
; This part deletes tuples that have been locked by the
; sites since the last refresh
;
for each relation in global snapshot
   where update-type = read/write
   key = key-attribute(relation)
   retrieve (relation.key, status.sent-to)
      where status.rights = locked
      begin
      for each site in status.sent-to
            except status.site-with-update-access begin
         mapped-relation = map(relation,site)
         delete from mapped-relation
          where key = relation.key
         end
      end
;
; The following operations are performed by the local
; SNAPS processors
;
; This operation commits all the updates,copies new data
;
for each relation in mapping-relation
   mapping = get-map(relation)
   send-result(mapping,
               "status.rights   = read/write or
                status.new-data = true       or
                status.modified = true",
               permanent)
   end
;
; set status.sent-to = false for those tuples marked
; deleted in the local relations
;
for each relation in mapped-relation
   mapping = map(relation)
   update mapping(status.sent-to(this-site) = false)
    where relation.status.rights = deleted
;
; delete tuples from the global snapshot if nobody has a
; copy of the data any more
;
for each relation in the global snapshot
   delete from relation
    where status.sent-to(all-sites) = false
;
; reset access rights
for each relation in the global snapshot
   update relation(status.rights = default(relation))
   end
```

The variable **mapped-relation** hold the names of the relations that are mapped onto the global snapshot for each site.

Algorithm to Manage the Allocation of Locks

This algorithm takes the relation name and qualification that defines the tuple that is wanted to be updated. The qualification is used that to access the status attribute(s) in the relation(s) to retrieve the locks. If the state of all the **access-rights** variable are all read/write then the request can be satisfied. The same mechanism as used in the local request to ensure that attempts to acquire the same locks by two different users do not conflict, is also used in this algorithm

```
function check-request(status-locks, qualification)

begin
   begin transaction
   retrieve into locks(status-locks)
      where qualification
   if not any(locks) where access-rights = locked
                        or access-rights = readonly then
      check-request = true
      update (status-locks = locked)
       where qualification
   else checked-request = false
      end
   end transaction
```

Algorithm to Manage User Enquiries

This algorithm basically passes the target-list and qualification to the database management system (DBMS) under the global snapshot controller. All the data that is required to perform the query has already been loaded into the database. When the query has been performed all the tuples that have their **access-rights** set to temporary and the **user-code** set to this user are deleted from the global snapshot relations.

```
procedure perform-query (target-list,qualification,
                         process-code)

begin
pass target-list + qualification to DBMS
for each relation in target-list and qualification
   delete from relation
    where status.rights = temporary
      and user-code = process-code
   end
end
```

Conclusion

Database snapshots are derived relations that provide an "as of" state of the database. Snapshots have been used in centralized databases to provide access to information that users require, or will tolerate, being out of date. They have been proposed as a method of access in distributed databases, rather than having to provide the concurrency control to keep the data up to date. Traditional snapshot mechanisms however are not very flexible.

The SNAPS system I propose provides the same features as traditional snapshots, with the following new extensions. The system is based on two level snapshots. The top level being a global snapshot of all the local databases, which is stored separately. The second level is a snapshot of the global snapshot, which is requested by a local site and stored in the local database as part of the local database.

To make the system more flexible, updatable snapshots are proposed. They have been implemented so that the overhead required is not significantly more than would be required for just a read only query on the global snapshot. Any changes to a local snapshot are made on the global snapshot and are then propagated to all the local snapshots when the global snapshot is refreshed.

The refresh mechanism I have developed uses the differential refresh approach. Only those tuples that have been added, modified, and or deleted are applied to the snapshot to refresh it.

The mapping system outlined provides a mechanism for mapping the schemata of local databases onto each other. It uses a global schema which all the local databases are mapped onto. This mechanism provides a way of implementing a global conceptual schema on top of the local conceptual schemata. One of the advantages of my system is that no other

site needs to know about the data that is stored at the other sites. It maps its query onto the global schema and then the query is processed by the global snapshot controller. This implies that there is a high degree of site autonomy.

Most approaches to distributed database design has involved integrating database in the same organisation. The system I propose is designed to be used with different organisations that are loosely joined. It is suitable for a database that is predominantly read only, and where limited update capability is required on only some relations in the database. It provides a high degree of autonomy to the different organisations. It is suitable when there is no large computer at each of the sites to co-ordinate the operations of the database.

This system has been designed for use in the radiology environment. It would be suitable for other loosely connected organisations such as libraries.

Appendices

A. Local Database Relation Status Attribute

This is the layout of the local database relation status attribute. This attribute is added to each local relation that is mapped onto the central database. This attribute is one byte long.

1. TUPLE TYPE — Size 1 bit
   - 0 local data
   - 1 snapshot

0 indicates that the tuple is local data, and not a snapshot.
1 indicates that the tuple is part of one or more snapshots.

2. CURRENT ACCESS RIGHTS — Size - 2 bits
   - 00 unknown
   - 01 read/write
   - 10 read only
   - 11 deleted

00 indicates that this relation has read/write access and that a lock will have to be requested from the central site before updates can be done on this tuple. No attempt has been made to acquire a lock before in this update cycle, so therefore the current status is unknown to this site.

01 indicates that a lock has been obtained from the central site for this tuple, and that this site has read/write access until the end of the update period.

10 indicates that this site only has read access, either because another site already acquired the read/write access, or the relation has no update access and can only be accessed on a read only basis.

11 indicates that the tuple has been deleted. It will be deleted when the central database is updated.

3. ROW IS NEW SINCE LAST UPDATE Size 1 bit
   0 data is not new
   1 data has been added since the last central update

0 indicates that this tuple has been in the local database before the last central database update.

1 indicates that this tuple has been added to the local database since the central database update. It is used when the central update is done, and then set to 0 to indicate that it has been copied to the central database.

4. ROW MODIFIED Size 1 bit
   0 not modified
   1 modified

0 indicates that this row has nor been modified since the last central database update.

1 indicates that the row has been modified, and that the changes will have to be sent tot the central database.

5. JUST COPIED Size 3 bit
   0 not just copied from central database
   1 to 7 just copied from the central database

0 indicates that the row has not just been copied over as part of a snapshot from the central database.

1 to 7 indicates that the row has been copied over as part of a snapshot. The system will use this fact to get the tuple identifiers for these tuples to put into the snapshot definition relation. The number indicates the code of the user that is performing the request.

B. Global Snapshot Relation Status Attribute

This attribute is added to each relation in the global snapshot. It is 46 bits long and each bit is coded in the following format. It would in practice be 48 bits long or 6 bytes per tuple.

1. ACCESS RIGHTS — Size - 2 bits
   - 00 lock acquired
   - 01 read/write
   - 10 read only
   - 11 temporary

00 indicates that this tuple has had a lock put on it by a site, and can not be updated by another site. This also indicates that this tuple has been modified.

01 indicates that this tuple has read/write access and no lock has been acquired on it yet during this update period.

10 indicates that this tuple can be read only but not updated.

11 is set if the tuple is only in the relation to perform a join operation with another relation.

2. SENT TO SITE — Size - 32 bits
   - 1 - 31 Site code

Each bit is set if this tuple has been sent to this site already. This is used to make sure that no tuples are sent unnecessarily, and also if they have been updated, the changes can be sent to each site when the daily update is done.

3. SITE WITH UPDATE ACCESS — Size - 6 bits
   - 0 No site has obtained this lock
   - 1 to 31 Site code

0 indicates that no site has obtained a lock for this tuple.

1 to 31 indicates which site has obtained the lock on this tuple.

4. USER CODE Size - 6 bits

0	Nobody accessing this tuple
1 to 31	Code of user using this tuple

0 indicates that no user is at present accessing this tuple.

1 to 31 indicate the code of a user that is accessing this tuple, so that the user can be identified if two or more users are accessing this relation at the same time. When a join operation is being performed the temporary status is also set. If more than one user from a site is copying this tuple to the local database, then another user from the same site will be prohibited from doing the same as this field is check when a copy is made.

C. Snapshot Description Relation

There is a **snapshot-description** relation which contains the snapshot number, the local relation name, and the list of tuple identifiers that make up this snapshot.

Each local relation has an extra attribute, which contains information such as tuple type, access rights, and new tuple flags. If the tuple is part of a snapshot then there is an tuple in this **snapshot-description** relation which contains the database system defined tuple identifier and the snapshot number and the local relation name as the key. The number of days since last access variable is used to determine if the snapshot can be deleted from the local database through lack of use.

1. TUPLE IDENTIFIER Size 4 bytes

The tuple identifier was 4 bytes in the Ingres system, though this value depends on the database management system being used.

2. SNAPSHOT NUMBER Size 2 bytes

This the system defined snapshot number, giving a range of 1 to 65535.

3. LOCAL RELATION NAME Size 12 bytes

In the Ingres system, the maximum size of a relation name is 12 characters, though this depends on the database management system being used.

4. NUMBER OF DAYS SINCE LAST ACCESSED Size 1 byte
   - 0 row locked into local database
   - 1 to 256 number of days since last accessed

0 indicates that this row is locked into the local database, and is not to be deleted.

1 to 256 indicates the number of days since this row was last accessed. A snapshot is held for up to a maximum of 256 days before it is deleted. This is used to delete unused snapshots from the local databases. The user at each site can set a value that suits the need at this site, and does not fill up the local database.

D. Snapshot definition relation

This is a relation that defines the snapshot of the central database. It includes the snapshot number, the local table name, and the qualification that is used to define the snapshot.

1. SNAPSHOT NUMBER Size 2 bytes
   1 - 65,535 Snapshot number

   Each snapshot has a unique number, which is used to identify which rows have been added to a local table as a snapshot.

2. TARGET LIST Size 80 bytes
   Snapshot target list

   String of all the attributes and relations in the target list.

3. QUALIFICATION Size 80 bytes
   String of the query qualification for snapshot

   Any valid qualification that returns rows for the local table name above. This size is arbitrary, and could be increased or decreased according to the system requirements.

E. Communication Options for Radiology System

Each local site is running the local area network(LAN), G/Net, for communications within a hospital or private surgery. The global snapshot can be located on a separate network, or connected to a local site. To connect these local sites to the global snapshot, a wide area communication system is required.

Telecom is the only supplier of wide area communications in New Zealand. They offer the following three systems that would be suitable, and compatible with the G/X25 product. They are leased lines, the packet switching network, PacNet, and the digital data network,DDN. Lines of speed 2400 bits per second (bps) or higher are considered, as anything slower is not considered suitable for file access operations.

First it was thought that it would be necessary to develop a communications and file system on top of the communications network. It was found that there is a system already available to do this, G/Net X25, that allows access to other G/Net LANs using X.25 as its communication protocol. This product also comes with a remote file server, allowing access to files on other LANs. Specifications of G/Net X25 can be found in Appendix F. This system can be used over the three types of communication system mentioned above.

Each option will be discussed with the following objectives in mind:

- Minimize installation, and usage costs
- Ease of implementation
- Flexibility of operation, and expansion

The installation costs, and the costs for 1 year's operation have been calculated and are used as a comparison between the three options. The Telecom charges as of March 1987, can be found in Appendix G.

All the costs calculated for the three options include modem hire. An alternative to hiring Telecom modems is to purchase them. This option should be considered when the long term status of the network has been determined.

The conclusions drawn from the comparison of the three options, where recommendations are made for the Nelson system, and the Christchurch system.

Leased Lines

This option would require one leased line between each local site and the central database site. A G/Net X25 board is required at the central database site and local site for each line to the central database site. Each board requires one expansion slot in a computer to act as a host.

Any new site will require a new board in the central site, and one in the new site itself. The cost for one site is therefore going to be the cost of two leased line connections.

Summary of costs of installation, and running costs for one year.

Installation	2400 bps	4800 bps	9600 bps
Connection (2 sites)	572.00	572.00	572.00
G/Net X.25 (2 boards)	11,990.00	11,990.00	11,990.00
Total Installation	12,562.00	12,562.00	12,562.00
Running costs for 2 months			
Modem rental (2 modems)	242.00	550.00	700.00
Distance charge - 5km	115.50	115.50	115.50
Distance charge - 10km	231.00	231.00	231.00
Total - 5km - 2 months	357.50	665.50	815.50

Total - 5km - 1 year	2,145.00	3,993.00	4,893.00
Total - 10km - 2 months	473.00	781.00	931.00
Total - 10km - 1 year	2,838.00	4,686.00	5,586.00

The capital outlay and installations costs for leased lines is expensive. This method is inflexible and any modification or addition to the network is expensive and requires large numbers of leased lines into the central database site.

Packet Switching Network

This option requires a packet switching network (PacNet) connection at the central database site and one at each local site. Each new site requires a connection at that site, and no new connection at the central database site. The costs are calculated for one site, as there is only one connection for each local site. The central cost sites are the same as for the local site.

As it is important that medical information be secure, all sites will be on a closed user group within PacNet. This ensures that those sites can only be accessed by other sites in this group. There is the capacity for encryption with in the X.25 specifications that could be used as well.

In calculating the costs, several estimates have had to be made. First, the costs are the same for each speed. Second, the connection time will be about 30 minutes per day. This is a small component of costs, and if it proves to be actually more than 30 minutes, it will not alter the costs significantly.

The usage charges will be based on the number of calls or queries, and the estimated amount of data transmitted per day to update the central database. The number of calls will be estimated at 100 per day at 6 segments per call. The number of segments transmitted while updating the central database is estimated at 100. To calculate for a month, a figure of 22 working days per month will be used.

Summary of installation costs and running costs for one site for one year.

Installation	2400 bps	4800 bps	9600 bps
Connection	616.00	616.00	616.00
Closed User Group	16.50	16.50	16.50
G/Net X25 board	5,995.00	5,995.00	5,995.00
Total Installation	6,627.50	6,627.50	6,627.50
Transmission Costs for 1 month			
Number of segments			
General queries	600		
Central update	100		
Total segments / day		700	
Transmission		0.85	
Connection time		0.33	
Total costs /per day		1.18	
Total Transmission - 1 month		25.96	
Running Costs	2400 bps	4800 bps	9600 bps
Connection	346.50	506.00	643.50
Closed User Group	2.42	2.42	2.42
Transmission	25.96	25.96	25.96
Total Running - 1 month	374.88	534.38	671.88
Total Running - 1 year	4,498.56	6,412.56	8,062.56

This option is the most simple to implement and extend. Any additional site can be added to the network by just installing a PACNET connection and a G/Net X25 board. No modification to any other site is needed.

The running costs are dominated by the connection cost and the actual transmission charges are only 8% of the total costs. If the transmission charges have been underestimated, then they are not likely to increase significantly.

Digital Data Network

This option requires one G/Net X25 board at each site, and one at the central site. This option is similar to the packet switching network option in ease of connection and modification. All costs are worked out as for one site.

Each new site requires a connection to the digital data network, and a G/Net X25 board in the back of a PC.

The transmission charges are at a fixed rate for 2 months, and do not depend on how much the network is used.

Summary of installation costs and running costs for one site for one year

Installation	2400 bps	4800 bps	9600 bps
G/Net X25 board	5,995.00	5,995.00	5,995.00
Installation	304.00	304.00	304.00
Total Installation	6,299.00	6,299.00	6,299.00
Running Costs - 2 months			
Connection	1,056.00	1,408.00	1,892.00
Transmission	66.00	143.00	308.00
Total - 2 months	1,122.00	1,551.00	2,200.00
Total - 1 year	6,732.00	9,306.00	13,200.00

Like the packet switching network, the digital data network has all the advantages of flexibility, and ease of implementation. The digital data network is essentially a leased line network with only one connection at each site, but is more expensive in all respects compared to the packet switching network.

F. Specification and Costs of G/X25 Net Gateway

G/X25 NET is a hardware software product providing G/NET LAN users shared access to remote LAN's, packet based mainframes and PC's through public data networks (PDN's) or point to point trunk lines.

The hardware component of G/X25 NET is a wide area network interface module Z80 (WNIM-Z80) - an intelligent communications controller that occupies one slot of the LAN PC workstation, and supports line speeds up to 19.2kbps. The communication functions are transparent to the workstation user. A synchronous modem is required.

The software components of G/X25 NET are the Interactive Personal Computer (IPC), that is an asynchronous PAD conforming to CCITT X.3 / X.28 / X.29 specifications, and the Remote File Server (RFS) that is the special interface that allows the Netware/G file server to be accessed from LAN's connected by wide area networks and local area networks.

G/X25 NET with 2 port WNIM-Z80, with RS232 Interface, Software, and manuals for G/NET PLUS LAN's	5995.00

G. Costs of Telecom's Telecommunication Costs

These are the Telecom charges as of March 1987, that were used to calculate the costs of running the various types of network considered for the Christchurch and Nelson communication networks for their radiology systems. All line speeds are in bits per second (bps), and all cost include the Government Service Tax (GST).

Packet Switching Network Costs

Network Connection	Line speed	Installation charge	Monthly rental
	2400	616.00	346.50
	4800	616.00	506.00
	9600	616.00	643.50

	Monthly rental
Closed User Group Service	
For each connection (all speeds)	2.42
Usage Charges	
Connection	0.66 / hour
Usage	1.21 / kilosegment (normal)[1]
	0.82 / kilosegment (off peak)

Digital Data Network Costs

Network Connection	Line speed	Installation charge	Monthly rental	
	2400	304.00	1,056.00	(836.00)
	4800	304.00	1,408.00	(1001.00)
	9600	304.00	1,892.00	(1276.00)

Usage Charges	Line speed	Monthly rental
	2400	66.00
	4800	143.00

[1] A kilosegment is 64,000 bytes. A segment is 64 bytes, and there is a minimum charge of six segments per message.

9600		308.00

In the Digital Data Network there is a central or controlling site. Costs for the operation are slightly less than a normal site, and are shown in brackets where they are different.

Leased Line Costs

	Line speed	Installation charge	Monthly rental
Line Rental			
	2400,4800,9600	286.00	23.10 / km

	Line speed	Installation charge	Monthly rental
Modem rental			
	2400	90.20	121.00
	4800	90.20	275.00
	9600	90.20	350.00

H. Schema of the Local Databases

The schema a defined below was designed by David Tripp and Alaistair Kenworthy. Some of the definitions of the relations have not been included as they have no relevance to this thesis, though their names have been include to show the general structure of the local databases. The relations that are shown are the one which are mapped onto the global snapshot.

The attributes and keys are defined in the following format : i = integer, c = character, and f = floating point number. The number associated with these code indicates the size in bytes.

Doctor

This relation stored the information about the doctors, radiologists, and radiographers. The unique key of this relation is **code**.

code	i4	title	c2
initials	c4	surname	c22
add1	c20	add2	c20
add3	c20	phone	c10

Out-Going Request

This relation stores requests for x-rays from this site. The unique key for this relation is all the attributes.

to_site	c4	from_site	c4
pno	i4	date	i4

Patient

This relations stores all the patient details. The unique key of this relation is **pno**. The relation has another unique key in **ad_no**.

pno	i4	ad_no	c7

title	c4	surname	c22
first_names	c24	dob	i4
sex	c1	race	c4
add1	c20	add2	c20
add3	c20	post_mortem	c1
dod	i4	vip	c1
contrast_sen	c1	gp	i2
no_of_visits	i2	film_locate	c8
perm_temp	c1	old_card	c1
dont_destroy	c1	films_in_mus	c1
account_due	c1	med_insure	c1

Radiological Services Bureau Codes

This relation stores the the Radiological Services Bureau codes that are used as the basis of calculating the health department rebate, and also in determining the type of x-ray performed. The unique key of this relation is the attribute **rsb_code**.

rsb_code	c3	description	c40
time_req	i2	mayo_units	i2
exam_total_f	f4	rsb_rebate	f4
code_1	c3	code_2	c3
code_3	c3		

Sites

This relation stores the information about the different sites. The unique key of this relation is **code**.

code	c4	int_code	i2
name	c30	this_site	c1
add1	c25	add2	c25
add3	c25	phone	c10
discount	c3	dept_vno	c12
gst_no	c15	acc_no	c15
label_addr	c5	recpt_addr	c5
inv_addr	c5	print_labels	c1
print_rec	c1	print_inv	c1

skip_rep_lin	i2	exit_recpt	i2
exit_radio	i2	no_labels	i2

Visit

This relation stores all the information concerning a patients visit, except the visit report, which is stored separately in the relation **visit_rep**. The unique key for this relation is the combination of **vno** and **pno**.

vno	i4	pno	i4
pvno	i2	dept_vno	c8
date	i4	time	i2
pat_location	c8	pat_type	c3
refer_doctor	i4	hosp_consult	i4
description	c30	rsb1	c3
rsb2	c3	rsb3	c3
rsb4	c3	rsb5	c3
rsb6	c3	roe1	f4
roe2	f4	roe3	f4
roe4	f4	radgrapher	c4
radiation_do	i4	rad_report	c4
rad_verify	c4	mayo_units	i2
films_destry	c1	teaching_use	c1
rep_name	c8	compressed	c1

Visit Report

This relation stores the actual report on the x-rays taken. The key of this relation is the combination of **vno** and **line_no**.

vno	i4	line_no	i2
line	c80		

The following relations are stored in the local databases but are the ones that are not mapped on to the global snapshot for one of the following reasons :

- local to that site
- restricted for confidentiality reasons,

- data stored in the relations is commercially sensitive

ACC Accounts	ACC Exam	Account Summary
Extra Reports	Free Patients	GST
Hospital Summary	Invoice	Invoice Payment
Incoming Request	Patient Comment	Patient Times
RSB Account	RSB Exam	Staff

I. Schema of the Global Snapshot

This appendix show the mapping of the local database onto the global snapshot. The schema was developed with the aid of Dr Kennedy. [KENN85]. This reflects his needs, but others might in the future decide that they want more or less of the local database mapped onto the global snapshot.

An attribute in the local database is not mapped onto the global snapshot if

- it is local only to that site
- has different values in each of the local databases
- for security and commercial reasons.

Doctor Relation

This relation maps exactly on to the local database equivalent **doctor.**

code	i4	title	c2
initials	c4	surname	c22
add1	c20	add2	c20
add3	c20	phone	c12

Out going Request Relation

This relation maps exactly onto its local database equivalent out_go_r.

to_site	c4	from_site	c4
pno	i4	date	i4

Patient Relation

A number of fields in the patient relation have not been mapped onto the global snapshot.

pno	i4	ad_no	c7

title	c4	surname	c22
first_names	c24	dob	i4
sex	c1	race	c4
add1	c20	add2	c20
add3	c20	post_mortem	c1
dod	i4	contrast_sen	c1
gp	i4		

Site Relation

This relation also has a number of attributes that are not mapped onto the global snapshot.

code	c4	int_code	i2
name	c30	add1	c25
add2	c25	add3	c25
phone	c10		

Visit Relation

A number of the attributes in the visit relation are no important enough to be stored in the global snapshot.

vno	i4	pno	i4
date	i2	time	i2
description	c30	pat_location	c4
pat_type	c3	refer_doctor	i4
hosp_consult	i4	reo1	f4
reo2	f4	reo3	f4
reo4	f4		

Visit Report Relation

This relation maps directly onto the local database equivalent **visit_rep**.

vno	i4	line_no	i2
line	c80		

References

[ABID80] Abida M.E. and Lindsay B.G. : *Database Snapshots* : Proceedings of the 7th Conference on Very Large Databases : 1980.

[ABID81] Abida M.E. : *Derived Relations : A Unified Mechanism for Views, Snapshots, and Distributed Data* : Proceedings of the 8th Conference on Very Large Databases : 1981.

[APER83] Apers P.M.G : *Query Processing and Data Allocation in Distributed Database Systems* : Mathematical Centre, Amsterdam : 1983.

[ASTR76] Astrahan M., et al : *System R : Relational Approach to Database Management* : ACM Transactions on Database Systems, Vol. 1, No. 2 : 1976.

[BADA80] Badal D.Z. : *On the Degree of Concurrency provided by Concurrency Control Mechanisms for Distributed Databases* : Proceedings of International Symposium on DDBs : North-Holland : 1980.[1]

[BANC80] Bancilhon F. : *Supporting View Updates in Relational Databases* : Proceedings of IFIP Working Conference on DBA : North-Holland : 1979.[2]

[BERN80a] Bernstein P.A., Shipman D.W. and Rohnie J.B. : *Concurrency Control in a System for Distributed Databases (SDD-1)* : ACM Transactions on Database Systems, Vol. 5, No. 1 : 1980.[3]

[BERN80b] Bernstein P.A., Shipman D.W. and Rohnie J.B. : *The Correctness of Concurrency Control Mechanism in a System for Distributed Databases (SDD-1)* : ACM Transactions on Database Systems, Vol. 5, No. 1 : 1980.

[BERN81] Bernstein P.A. and Chiu D.W. : *Using Semi-Joins to Solve Relational Queries* : Journal of the ACM, Vol. 28, No. 1 : 1981.

[1] DDBs is an acronym for Distributed Databases
[2] DBA is an acronym for Database Architecture
[3] ACM is an acronym for Association of Computer Machinery

[BERN83] Bernstein P.A. and Goodman N. : *Multiversion Concurrency Control - Theory and Algorithms* : ACM Transactions on Database Systems, Vol. 8, No. 4 : 1983.

[BERN84] Bernstein P.A. and Goodman N. : *An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases* : ACM Transactions on Database Systems, Vol. 9, No. 4 : 1984.

[CELL80] Cellary W. and Meyer D. : *A Multi-query Approach to Distributed Processing in a Relational Distributed Database Management System* : Proceedings of International Symposium on DDBs : North-Holland : 1980.

[CERI83] Ceri S. and Pelagatti G. : *Correctness of Query Execution Strategies in Distributed Databases* : ACM Transactions on Database Systems, Vol. 8, No. 4 : 1983.

[CERI84] Ceri S. and Pelagatti G. : *Distributed Databases : Principles and Systems* : McGraw-Hill : 1984.

[CODD70] Codd E.F. : *A Relational Model of Data for Large Shared Banks* : Communications of the ACM, Vol. 13, No. 6 : 1970.

[CODD79] Codd E.F. : *Extending the Database Relational Model to Capture More Meaning* : ACM Transactions of Database Systems, Vol. 4, No. 4 : 1979.

[CONC86] Concept Data Systems Ltd : *Discussion held with the management and employees of this company* : 1986, and 1987.

[COOP86a] Cooper R.E.M. : *Snapviews - A Rational Approach to Snapshots* : Unpublished Paper, Computer Science Department, University of Canterbury : 1986.

[COOP86b] Cooper R.E.M. : *Snapshots in Mimer* : Unpublished Paper, Computer Science Department, University of Canterbury : 1986.

[DATE86] Date C.J. : *An Introduction to Database Systems (Vol. 1 ,4th Edition)* : Addison-Wesley USA : 1986.

[DAVI84] Davidson S. B. : *Optimism and Consistency in Partitioned Distributed Database Systems* : ACM Transactions on Database Systems, Vol. 9, No. 3 : 1984.

[EPST79] Epstein R., Stonebraker M.R., and Wong E. : *Distributed Query Processing in a Relational Database System* : Proceedings of the ACM-SIGMOD : 1979.

[ESWA76] Eswaran K., et al : *On the Notions of Consistency and Predicate Locks in a Relational Database System* : Communications of the ACM : Vol. 19, No. 11 : 1976.

[FELT85] Feltham Dr. : *Personal discussion and in meetings with other parties* : 1985, 1986 and 1987.

[GARC79] Garcia-Molina H. : *A Concurrency Control Mechanism for Distributed Database which uses Centralized Locking Controllers* : Proceedings of 4th Int. Workshop on Distributed Data Management and Computer Networks : 1979.

[GARC83] Garcia-Molina H. : *Using Semantic Knowledge for Transaction Processing in a Distributed Database* : ACM Transactions on Database Systems, Vol. 8, No. 2 : 1983.

[GATE85] Gateway Communications Inc. : *Product specification of G/X25* : 1985.

[GATE87] Gateway Communications Inc. : *Product specification of G/Net* : 1987.

[GOOD81] Goodman N., et al : *Query Processing in SDD-1 : A System for Distributed Databases* : ACM Transactions on Database Systems, Vol. 6, No. 4. 1981.

[INFO86] *Informix Users Manual* : 1986.

[JOSE86] Joseph T.A. and Birman K.P. : *Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems* : ACM Transactions on Computer Systems Vol. 4, No. 1 : 1986 :

[KALA86] Kalash J. : *Ingres Reference Manual Version 8* : 1986.

[KENN85] Kennedy Dr : *Personal discussions and in meetings with other parties* : 1985, 1986 and 1987.

[KERN78] Kernighan B.W. and Ritchie D.M. : *The C Programming Language* : Prentice-Hall, New Jersey : 1978.

[KUNG81] Kung H.T. and Robinson J.T. : *On Optimistic Methods of Concurrency Control* : ACM Transactions on Database Systems, Vol. 6, No. 2 : 1981.

[LIND86] Lindsay B.G., et al : *A Snapshot Differential Refresh Algorithm* : ACM Special Interest Group Management of Data, Vol. 15, No. 2 : 1986.

[MAHM76] Mahmoud S.A. and Riordon J.S. : *Optimal Allocation of Resources in Distributed Information Networks* : ACM Transactions on Database Systems, Vol. 1, No. 1 : 1976.

[MAHM79] Mahmoud S.A., Riordon J.S. and Toth K.C. : *Distributed Database Partitioning and Query Processing* : Proceedings of the IFIP Working Conference on DBA : North-Holland : 1979.

[MICR87] Micro Networks Ltd. : *Letter providing costs of G/Net LAN and G/X25 systems* : 1987.

[MILE80] Milenkovic M. : *Synchronization of concurrent updates in redundant distributed databases* : Proceedings of International Symposium on DDBs : North-Holland : 1980.

[MUNZ79] Manz R. : *Gross Architecture of the Distributed Database System VDN* : Proceedings of the IFIP Working Conference on DBA : North-Holland : 1979.

[PAPA84] Papadimitriou C.H. and Kanellakis P.C. : *On Concurrency Control by Multiple Versions* : ACM Transactions on Database Systems, Vol. 9, No. 1 : 1984.

[POPE80] Popescu-Zeletin R. and Weber H. : *Some considerations about Distributed Databases on Public Networks* : Proceedings of International Symposium on DDBs 1980 : North-Holland : 1980.

[SACC85] Sacca D. and Wiederhold G. : *Database Partitioning in a Cluster of Processors* : ACM Transactions on Database Systems, Vol. 10, No. 1 : 1985.

[SCHL81] Schlagter G. : *Optimistic Methods of Concurrency Control in Distributed Databases* : Proceedings of the 7th Conference on Very Large Databases : 1981.

[SEGE86] Segev A. : *Optimisation of Join Operations in Horizontally Partitioned Database Systems* : ACM Transactions on Database Systems, Vol. 11, No. 1 : 1986.

[STON76] Stonebraker M.R., et al : *The Design and Implementation of INGRES* : ACM Transactions on Database Systems, Vol. 1, No. 3 : 1976.

[STON79] Stonebraker M.R. : *Concurrency Control and consistency of Multiple Copies of Data in Distributed Ingres* : IEEE Transaction on Software Engineering, Vol. 5, No. 3 : 1979.

[TAY85] Tay Y.C., Goodman N. and Suri R : *Locking Performance in Centralized Databases* : ACM Transactions on Database Systems, Vol. 10, No. 4 : 1985.

[THOM79] Thomas R.W. : *A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases* : ACM Transactions on Database Systems, Vol. 4, No. 2 : 1979.

[TRIP86] Tripp D. and Kenworthy A. : *Discussion Paper on the proposed Radiology System* : Unpublished Paper : 1987.

[ULLM82] Ullman J.D. : *Principles of Database Systems (2nd ed.)* : Computer Science Press : 1982.

[WONG76] Wong E. and Youssefi K. : *Decomposition - A Strategy for Query Processing* : ACM Transactions on Database Systems, Vol. 1, No.3 : 1976.

[YU84] Yu C.T and Chang C.C. : *Distributed Query Processing* : ACM Computing Surveys Vol. 16, No. 4 : 1984.